Using CIMPLE V2 [1]

A Practical Guide to Developing CIM Providers

Michael Brasher (*m.brasher@inovadevelopment.com*
Karl Schopmeyer (*k.schopmeyer@inovadevelopment.com*

Wednesday 8th December, 2010

[1]This version of the User Guide corresponds to CIMPLE Version 2.

Using CIMPLE, Version 2.2

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This guide explains how to use `CIMPLE` to develop CIM providers. We assume you already know about CIM and WBEM and that you are looking for a better way to build providers. As you will see, `CIMPLE` makes provider development faster and easier and the end-product is more reliable and maintainable.

1.1 Who Are We?

The authors are founders of the OpenPegasus project. Karl is the project manager of OpenPegasus and Michael was the original developer and first architect. While working with the OpenPegasus community, we repeatedly see programmers struggle with provider development. Building a provider is a painstaking and costly activity, which is why we started the `CIMPLE` project. We welcome you to the community of developers who are using `CIMPLE` to develop providers with less effort and less cost.

1.2 What Is `CIMPLE`?

`CIMPLE` is an open-source environment for building CIM providers that are compatible with several CIM server implementations. `CIMPLE` providers function transparently under several prominent provider interfaces including:

- OpenGroup CMPI Specification Version 2
- OpenPegasus C++ Provider Interface
- OpenWBEM C++ Provider Interface (not maintained today)
- Microsoft WMI Provider Interface (starting with `CIMPLE` 2.0)

With `CIMPLE`, developers can produce one provider that works with multiple CIM server implementations and multiple operating systems

Unlike traditional provider interfaces, `CIMPLE` translates CIM classes into concrete C++ classes. *Concrete classes* substantially reduce code complexity and improve type safety.

1.3 Why Use `CIMPLE`?

Developers use `CIMPLE` because it offers four major advantages over conventional provider interfaces.

- Substantially reduces development effort.
- Promotes type-safety and program correctness.
- Produces small-footprint providers.
- Supports multiple provider interfaces.
- Interoperates with several CIM servers.

Each of these is discussed below.

1.3.1 Reducing Development Effort

`CIMPLE` reduces development effort in two ways. First, providers are easier to develop in the first place, due to code generation, reduced code complexity, type safety, and operation reduction (see section 1.4). Second, you can develop a single provider that works transparently with multiple provider interfaces (see section 1.3.3).

1.3.2 Developing Small-Footprint Providers

`CIMPLE` is ideal for developing providers with a small footprint. A provider's footprint refers to the total object size of the provider library. `CIMPLE` providers are comparable in size to CMPI providers and many times smaller than OpenPegasus providers.

1.3.3 Supporting Multiple Provider Interfaces

Providers developed with `CIMPLE` function transparently under the following different provider interfaces:

- OpenGroup CMPI Specification Version 2

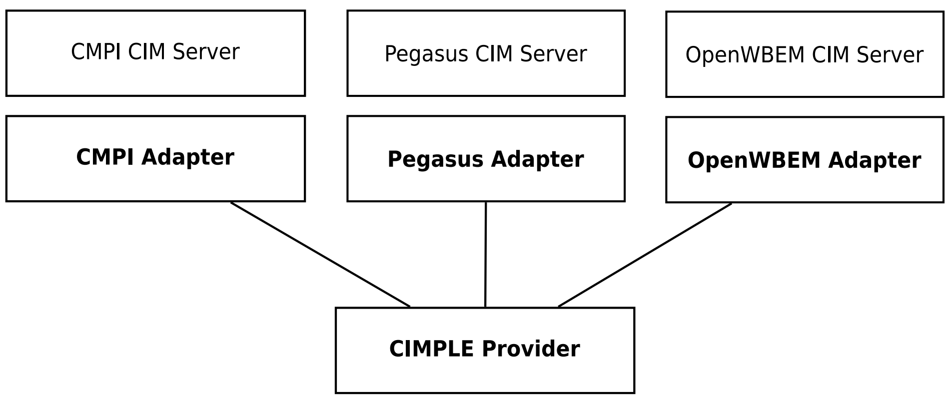
- OpenPegasus C++ Provider Interface
- OpenWBEM C++ Provider Interface
- Microsoft WMI Provider Interface

CIMPLE supplies an *adapter* for each of these interfaces. Configuring a CIMPLE provider for a provider interface is a simple matter of linking with the corresponding adapter. No source code changes are necessary.

1.3.4 Interoperating With Multiple CIM Servers

With CIMPLE, you can develop a single provider that works with different CIM servers. This is achieved through the use of provider adapters described in the previous section. Figure 1.1 shows a CIMPLE provider functioning under three kinds of CIM servers (CMPI, OpenPegasus, and OpenWBEM). Note that CIMPLE providers work with all CMPI-enabled servers.

Figure 1.1: Multi CIM Server Support



1.4 Major Simplifications

CIMPLE offers four major simplifications over conventional provider development technologies.

- concrete classes
- provider skeleton generation

- extrinsic method stub generation
- provider operation reduction
- automated provider registration

Each is described in the following subsections.

1.4.1 Concrete Classes

With `CIMPLE`, developers work with *concrete classes* generated directly from MOF class definitions. Concrete classes substantially reduce code complexity and bring CIM classes under the scrutiny of the C++ static type checking facility. For example, consider the following MOF definition for the class `President`.

```
class President
{
    [Key] uint32 Number;
    string First;
    string Last;
};
```

The `CIMPLE` `genclass` command generates a C++ class from this definition. The following snippet creates an instance of the generated `President` class .

```
President* inst = President::create();
inst->Number.set(1);
inst->First.set("George");
inst->Last.set("Washington");
```

Creating the same instance in or is considerably more difficult. See section A.1 for equivalent OpenPegasus and CMPI snippets. The table below summarizes the complexity of each implementation.

	Lines	**Characters**
CIMPLE	4	126
OpenPegasus	18	502
CMPI	57	969

In addition to the obvious reduction in code complexity, `CIMPLE` has other advantages as well.

- Type safety
- Smaller code size
- Better performance

Typical errors encountered with conventional provider interfaces like OpenPegasus and CMPI include the following.

- Misspelled or unknown properties names
- Misspelled or unknown classes names
- Wrong parameter types

With conventional providers, these errors are detected only at run time, whereas with `CIMPLE` they are detected at compile time.

1.4.2 Provider Skeleton Generation

`CIMPLE` generates provider skeletons automatically from MOF class definitions. The following command, for example, generates provider skeletons for the `President` class, defined above.

```
$ genprov President
Created President_Provider.h
Created President_Provider.cpp
```

This skeleton includes the provider class declaration and stubs for each of the `President` provider methods. Once the skeleton is generated, developing a provider is a matter of implementing the stubs. The generated source code for this example is included in appendix B.

Sometimes `genprov` is used to “patch” an existing provider for which provider code has already been written. This is needed in two situations.

- The MOF class definition changed (a property or extrinsic method was added, deleted, or changed).
- `CIMPLE` changed an intrinsic method signature (very rare).

1.4.3 Extrinsic Method Stub Generation

`CIMPLE` makes it much easier to implement extrinsic methods by generating a stub for each method in the CIM class. For example, consider the following MOF class definition.

```
class Adder
{
    real64 add(real64 x, real64 y);
};
```

The `add()` method returns the sum of its two parameters. The `CIMPLE genprov` command generates a stub for the add method as shown below.

```
Invoke_Method_Status Adder_Provider::add(
    const Adder* self,
    const Property<real64>& x,
    const Property<real64>& y,
    Property<real64>& return_value)
{
    return INVOKE_METHOD_UNSUPPORTED;
}
```

Inserting the code that defines behavior as below finishes the implementation.

```
Invoke_Method_Status Adder_Provider::add(
    const Adder* self,
    const Property<real64>& x,
    const Property<real64>& y,
    Property<real64>& return_value)
{
    return_value.set(x.value + y.value);
    return INVOKE_METHOD_OK;
}
```

Implementing the stub required two lines of original code, setting the return value and returning with the OK status. Compare this with the 102 lines required by the CMPI implementation shown in section A.2.

1.4.4 Provider Operation Reduction

Another way `CIMPLE` simplifies provider development is by reducing the number of provider operations that must be implemented. The following operations have been eliminated, either because they are special cases of other operations or they can be automated:

- **enumerate-instance-names** – special case of **enumerate-instances**.
- **associators** – implemented using **associator-names**.
- **reference-names** – special case of **references**.
- **create-subscription** – automated by adapter.
- **modify-subscription** – automated by adapter.
- **delete-subscription** – automated by adapter.

Additionally, the following operations are optional, since they can be implemented by the `CIMPLE` environment in terms of other operations. For these operations to be implemented by the `CIMPLE` environment itself, the provider writer need only to implement `enumerate_instances` and and leave the generated skeleton for these other functions to return unsupported.

- **get-instance** – implemented with **enumerate-instances** if unsupported.
- **associators** – implemented with **enumerate-instances** if unsupported.
- **references-names** – implemented with **enumerate-instances** if unsupported.

The minimal set of operations that *must* be implemented by each of the provider types is shown in table 1.1 below. For example, many instances providers only need to implement **enumerate-instances** in order to implement a complete provider.

Table 1.1: **Minimal Provider Implementation**

Provider Type	**Required Operations**
Instance	**enumerate-instances**
Association	**enumerate-instances**
Method	**invoke-method**
Indication	**enable-indications**, **disable-indications**

1.4.5 Provider Module Generation

A provider module is the physical (typically shared library) software package that contains one or more providers. In `CIMPLE` each provider manages a single class and is generated with the `CIMPLE` utility `genprov`. The provider module file contains `CIMPLE` registration information for the provider and an entry point for the provider interfaces.

The choice of how many providers to incorporate into a single provider module is a developers choice depending on a lot of factors such as intercommunication between providers, the issues of distribution (distribute one or more shared libraries), commonality of lower level functions, etc.).

To build a provider module you simply execute `genmod` with the set of providers to be included in the module which produces a module.cpp output file that is the provider Module.

To build multiple providers into a single provider module, typically you build them all in the same directory and, create the module file with `genmod` and compile the whole set. The example Makefiles and the `genmake` utility that generates Makefiles for providers are based on this premis.

1.4.6 Provider Registration

Today provider registration has not been standardized in the DMTF CIM model. Therefore there is no single definition of provider registration and typically each CIM Server defines its own provider registration mechanisms. Since these processes are significantly different the registration process for each of the supported platforms is described independently in the following sections.

See section 3.8 for more detailed information on registration for different CIM Servers.

As an example, to register our provider for Pegasus we use the `CIMPLE regmod` utility that automates the entire process including:

- Building instances of the three classes required to register a provider for pegasus
- Installing these instances into the running server
- Copying the provider library to the location defined by the server

For example, the following command registers the providers contained in `libPresident.so`.

```
$ regmod libPresident.so
Using CMPI provider interface
Registering President_Provider (class President)
```

This is much easier than the manual approach, which involves writing MOF registration instances and compiling them with the OpenPegasus `cimmof` MOF compiler.

1.4.7 Provider Testing

Once our provider is installed and registered in Pegasus, we can test it easily with any number of client tools. For this description we chose the Pegasus client test tool `cimcli`—.

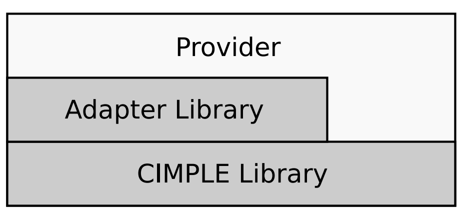
The commands below should demonstrate what information the CIM Server and the provider for the President class are returning.

```
$ cimcli ei President
. . . . Returns the instances generated by the provider
$ cimcli ni President
. . . . Returns the object paths generated by the provider
$ cimcli gi President.key=1
. . . . Returns the single instance with key = 1
```

At this point we have a complete provider that generates instances of our President class.

1.5 Architectural Overview

The architecture of a `CIMPLE` provider is simple. As shown in the figure below, a `CIMPLE` provider uses two libraries: the main `CIMPLE` library and one adapter library (CMPI, OpenPegasus, OpenWBEM, or WMI).



The provider library itself defines the appropriate entry points as part of the `module.cpp`, discussed in chapter 3.

The `CIMPLE` library provides the implementation of the `CIMPLE` functions which manage instances, provider operations, and provide a standard set of platform independent functions generally required for providers such as data handling, logging, thread management, etc.

The adapters provide the adaption of `CIMPLE` standard objects for instances to and from the objects required by the target CIM Server and interface with the Server.

Chapter 2

Installing CIMPLE

This chapter explains how to download, configure, build and install CIMPLE. Generally the procedure is as follows.

```
$ ./configure
$ make
$ make install
```

But be careful since this only builds CIMPLE standalone, without support for , , and . Please read section 2.3 to see how to configure support for these .

Currently CIMPLE supports Windows and several Linux platforms. The following is a complete list of supported targets.

- Linux IX86 32-bit
- Linux IX86 64-bit
- Linux S390 32-bit
- Linux S390X 64-bit
- Linux IA64
- Linux PPC 32-bit
- Linux PPC 64-bit
- Solaris Sparc 32-bit and 64-bit
- Solaris IX86 32-bit and 64-bit
- Windows IX86

Porting CIMPLE to POSIX platforms is relatively easy. Please contact us if you need CIMPLE ported to other platforms.

2.1 CIMPLE License

CIMPLE is licensed using the MIT license. To you this means that the license places minimal requirements on your use of CIMPLE.

In general, the only requirement is that you maintain our license and copywrite statement on the source code if you redistribute the CIMPLE source code.

CIMPLE places no licensing on the source code produced by CIMPLE.

2.2 Downloading

`CIMPLE` source distributions are available from http://cimple.org. We recommend downloading the latest release, which is always listed at the top of the downloads page (http://simplewbem.org/downloads.html). Source distributions are available as gzipped tar files and as zip files. For example, the `CIMPLE` 2.0.0 source distribution is available as `cimple-1.0.0.tar.gz` and `cimple-2.0.0.zip`.

We assume you know how to unpack a source distribution. Unpacking the `CIMPLE` distribution creates the *CIMPLE root directory*.

2.3 Configuring

To build `CIMPLE` standalone (without support for any of the provider adapters such as , , or), change to the `CIMPLE` root directory and type the following.

```
$ ./configure
```

The `configure` tool requires command-line options to support the various provider adapters such as CMPI, OpenPegasus, or OpenWBEM. To get a list of configure options type the following.

```
$ ./configure --help
```

To see how to configure `CIMPLE` for , , or , read the corresponding subsection below.

2.3.1 Configuring for CMPI

To configure `CIMPLE` to support CMPI, use this option.

```
--with-cmpi=DIR
```

`DIR` is the name of the directory that contains the standard CMPI header files (e.g., `cmpidt.h`, `cmpift.h`, `cmpimacs.h`). For example, the following tells `CIMPLE` that the standard CMPI headers are located under `/usr/include/cmpi`.

```
$ configure --with-cmpi=/usr/include/cmpi
```

This configuration builds `CIMPLE` and the CMPI adapter.

2.3.2 Configuring for OpenPegasus – RPM Distribution

In general, you can configure `CIMPLE` for !configure CIMPLE with the following option.

```
--with-pegasus=DIR
```

`DIR` is the name of the directory where `CIMPLE` expects to find the directories `bin`, `lib`, and `include` directories. You can specify different locations for `lib` and `include` with these options.

```
--with-pegasus-libdir=DIR
--with-pegasus-includedir=DIR
```

This configuration builds `CIMPLE`, the adapter adapter, and the `regmod` tool.

2.3.3 Configuring for OpenPegasus – Source Distribution

To build `CIMPLE` for use with an configure source distribution, use the following option in conjunction with the standard env vars environment variables.

```
--with-pegasus-env
```

With this option, the `configure` tool deduces the configuration options from the following !environment variables environment variables.

```
PEGASUS_HOME
PEGASUS_ROOT
PEGASUS_PLATFORM
PEGASUS_DEBUG
```

This is equivalent to configuring with the following configuring CIMPLE options.

```
--prefix=$PEGASUS_HOME
--libdir=$PEGASUS_HOME/lib
--with-pegasus=$PEGASUS_HOME
--with-pegasus-libdir=$PEGASUS_HOME/lib
--with-pegasus-includes=$PEGASUS_ROOT/src
--with-cmpi=$PEGASUS_ROOT/src/Pegasus/Provider/CMPI
```

This configuration builds `CIMPLE`, the OpenPegasus adapter, the CMPI adapter, and the `regmod` tool. `CIMPLE` is installed under the directory given by the environment variable .

2.3.4 Configuring for OpenWEBM

You can configure for !configure CIMPLE with the following option.

```
--with-openwebem=DIR
```

`DIR` is the name of the directory where `CIMPLE` expects to find the OpenWBEM `bin`, `lib`, and `include` directories. This configuration builds `CIMPLE`, and the OpenWBEM adapter.

2.3.5 Configuring for WMI

You can configure for !configure for with the following options:

```
--bindir=c:/windows/system32
--enable-wmi
```

The following command executed from the CIMPLE root directory would configure CIMPLE for creating a WMI provider.

```
C:\> configure.bat --bindir=c:/windows/system32 --enable-wmi
```

2.4 Building

CIMPLE is built by typing the make command from the CIMPLE root directory.

```
    $ make
```

This builds CIMPLE as configured above. You can check the build with the make Makefile command.

```
    $ make check
```

This runs a handful of unit tests to see if the resulting build is usable on your platform. It does not test the providers installed in a server.

2.5 Installing

To install CIMPLE, type the following command.

```
    $ make install
```

This installs the CIMPLE libraries into the locations selected by the configure tool. This does not install any providers generated with the CIMPLE utilities.

Chapter 3

Getting Started

In this chapter you will learn how to develop a simple instance provider. There are 8 steps to developing a CIMPLE provider.

1. Define the class.
2. Generate the class. (CIMPLE `genclass` utility)
3. Generate the provider. (CIMPLE `genprov` utility)
4. Generate the module file. (CIMPLE `genmod` utility)
5. Implement the skeleton.
6. Build the provider.
7. Register the provider. (CIMPLE `regmod` utility)
8. Test the provider.

Most steps are automated, taking only a couple of minutes to perform. The relevant automation tools are shown in parentheses above.

> `genproj`. The `genproj` tool, introduced by CIMPLE 1.0.0, runs `genclass`, `genprov`, and `genmod`, which reduces source code generation to a single step.

The provider featured in this chapter provides instances of the first three American presidents.

3.1 Defining the Class

First we define the `President` MOF class, placing it in a file called `repository.mof`. The Classes can be placed directly into the `repository.mof` or defined in their own classes and the `repository.mof` file used to reference the other mof files with the include pragma.

```
class President
{
    [Key] uint32 Number;
    string First;
    string Last;
};
```

This class has a single key property (`Number`) and two other properties (`First` and `Last`).

3.2 Generating the Class

Now we generate the C++ class from the MOF definition by typing the following command `genclass` in the directory that contains `repository.mof`. (You only need this file when defining new classes that are not in the CIM schema).

```
$ genclass -r President
Created President.h
Created President.cpp
created repository.h
Created repository.cpp
```

This command creates four files.

- `President.h` – the President class
- `President.cpp` – internal definitions used by CIMPLE
- `repository.h` – internal definitions used by CIMPLE
- `repository.cpp` – internal definitions used by CIMPLE

The two `.cpp` files must be included in the provider build (discussed in section 3.7). The `-r` option creates `repository.h` and `repository.cpp`. Remember that since you will never edit the generated class files, you can regenerate them whenever you change the MOF definition.

When you develop a multi-provider !multi-provider module, you should generate all the classes at once. For example, suppose your provider module provides these classes.

- President
- VicePresident
- VicePresidentAssociation

Then generate all three classes with a single execution of `genclass` as follows.

```
$ genclass -r President VicePresident VicePresidentAssociation
```

Also, remember to always use the `-r` option/indexgenclass!-r option.

3.3 Generating the Provider

The `genprov` tool generates provider skeletons. To generate a provider skeleton for the `President` class, type the following command from the directory that contains `repository.mof`.

```
$ genprov President
Created President_Provider.h
Created President_Provider.cpp
```

This command creates two files.

- President_Provider.h
- President_Provider.cpp

Appendix B includes a complete listing of the generated files.

The generated source file is a valid provider that provides zero instances of the `President` class, and responds with "NOT_SUPPORTED" to create, delete, and modify operation requests. In section 3.6, we extend this skeleton, by implementing the `enum_instances` and `get_instance` methods.

> **genprov patching**. CIMPLE 1.0.0 added a patching feature to `genprov`. If the the provider sources already exist, `genprov` patches the method signatures and inserts any new extrinsic methods. This allows you to modify the class definitions and regenerate classes, providers, and provider modules without losing existing implementation code.

3.4 Generating the Module

Next we show how to generate `module.cpp` for the President provider. This file contains `CIMPLE` registration information for the provider and an entry point for one of the following provider interfaces.

-
-
-
-

The `genmod` command generates `module.cpp` for our President provider.

```
    $ genmod President President
    Created module.cpp
```

The first argument (`President`) is the name of the module. The second argument (also `President`) is the name of the provider class.

Since the module file is regenerated whenever new providers are added to the module, it should never be edited. For example, adding a `VicePresident` provider to the module, requires regenerating the `module.cpp` with the following command.

```
{
    $ genmod -f President President VicePresident
    Created module.cpp
```

Later, in section 3.7, we show how to compile `module.cpp` together with all the other generated files to build a module library.

3.5 Generating a Provider Makefile

Next we show how to generate a `Makefile` for the President provider. This step is optional and creates a Makefile defined to be usable with . The created `Makefile` contains the setup for building the provider.

```
$ genmak -f President President.cpp VicePresident.cpp module \
            repository.cpp President_Provider.cpp VicePresident_Provider.cpp
Created Makefile
```

The `genmak` utility includes an option (-C) to allow creating the Makefile to generate cmpi providers and another (-f) to overwrite any existing Makefile.

This utility creates a Makefile of approximately the following form:

```
##
## Makefile generated by genmak version 2.0.5
##

TOP=/home/username/cimplework/pegasus/LINUX_X86_64_GNU/share/cimple
ROOT=.
BINDIR=.
LIBDIR=.
include $(TOP)/mak/config.mak

MODULE=1
SHARED_LIBRARY=President

##
## Define source files for compile and link
##

SOURCES += President.cpp
SOURCES += repository.cpp
SOURCES += module.cpp
SOURCES += President_Provider.cpp

##
## Module defined as Pegasus C++ interface
##
CIMPLE_PEGASUS_MODULE=1
DEFINES += -DCIMPLE_PEGASUS_MODULE

LIBRARIES += cimplepegadap
LIBRARIES += cimple

include $(TOP)/mak/rules.mak
```

3.6 Implementing the Skeleton

In this section we implement the provider skeleton, generated in section 3.3 (see appendix B for the listing). We implement a "read-only" provider, which invovles implementing these two stubs.

- `President_Provider::enum_instances`
- `President_Provider::get_instance`

Note that we do not have to implement `get_instance`. It will provide instances based on the implemented `enum_instances` if the generated stub is not modified.

The complete source for this provider is included in appendix C.

3.6.1 Implementing the `enum_instances` Stub

Earlier we used `genprov` to generate the following stub.

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

This method is called to service two CIM operations.

- **enumerate-instances**
- **enumerate-instance-names**

The following implementation provides a single instance of `President`.

```
Enum_Instances_Status President_Provider::enum_instances(
    const President* model,
    Enum_Instances_Handler<President>* handler)
{
    President* inst = President::create(true);
    inst->Number.set(1);
    inst->First.set("George");
    inst->Last.set("Washington");

    handler->handle(inst);

    return ENUM_INSTANCES_OK;
}
```

Lines 5 through 8 create and initialize the `President` instance. Line 10 sends the new instance to the requestor. It should be easy to see how to extend this to provide additional instances.

Notice that the implementation above ignores the `model` parameter. This parameter identifies the minimal set of required properties. That is, it indicates which properties the provider *must* provide. The following snippet checks whether the `First` property is required.

```
if (!model->First.null)
{
    // Property First is required.
}
```

There are two cases where a subset of properties is requested.

- `Enum_instances` is servicing an **enumerate-instances** request, in which the requestor selected properties with a property list.
- `Enum_instances` is servicing an **enumerate-instance-names** request, which requires only key properties.

A provider may safely ignore the `model` parameter and provide all properties instead. However, using the `model` improves performance by avoiding unecessary property value fetches.

The following snippet is a revision of the `enum_instances` implementation that utilizes the `model` parameter.

```
1  Enum_Instances_Status President_Provider::enum_instances(
2      const President* model,
3      Enum_Instances_Handler<President>* handler)
4  {
5      President* instance = President::create(true);
6      instance->Number.set(1);
7
8      if (!model->First.null)
9         instance->First.set("George");
10
11     if (!model->Last.null)
12         instance->Last.set("Washington");
13
14     handler->handle(instance);
15
16     return ENUM_INSTANCES_OK;
17 }
```

Line 5 creates a `President` instance whose properties are null by passing `true` to create. Line 8 checks whether the `First` property is required. Line 11 checks whether the `Second` property is required. It is unecessary to check the `Number` property since keys are always required.

The `enum_instances` method must return one of the following return codes. [1]

- `ENUM_INSTANCES_OK`
- `ENUM_INSTANCES_FAILED`
- `ENUM_INSTANCES_ACCESS_DENIED`,

Returning any other integer value causes a compilation error, since the return type is a C++ enumeration.

The CIMPLE environment automatically handles processing other parameters of the input request such as property lists without any modifications to the above code.

3.6.2 Implementing the `get_instance` Stub

Next we implement the `get_instance` method. Here is the stub generated by `genprov`.

[1]We have defined all legal return types for each operation type that are defined in the DMTF specification and NOT handled automatically by the CIMPLE environment. Note that the enumerations are specific to each operation and also that they are not the same list for all operations.

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

As it stands, the generated stub is a valid implementation of `get_instance`. Returning `GET_INSTANCE_UNSUPPORTED` causes the adapter to satisfy the request by calling the `enum_instances` method and then filtering the return for the instance defined by the model parameter. But providing a "proper" implementation of `get_instance` may improve performance. The following snippet provides a full implementation of the `get_instance` method.

```
 1  Get_Instance_Status President_Provider::get_instance(
 2      const President* model,
 3      President*& instance)
 4  {
 5      if (model->Number.value == 1)
 6      {
 7          instance = President::create(true);
 8          instance->Number.set(1);
 9          instance->First.set("George");
10          instance->Last.set("Washington");
11          return GET_INSTANCE_OK;
12      }
13
14      return GET_INSTANCE_NOT_FOUND;
15  }
```

The `model` parameter contains the keys for the requested instance. Line 5 checks to see whether `President.Number=1` has been requested. Lines 7 through 10 create and initialize the new instance. Line 11 returns `GET_INSTANCE_OK`. If the model does not match any known instance, the method should return `GET_INSTANCE_NOT_FOUND` (line 14). It should be fairly obvious how to extend `get_instance` to provide additional instances. The `enum_instances` method must return one of the following.

- `GET_INSTANCE_OK`

- GET_INSTANCE_FAILED
- GET_INSTANCE_UNSUPPORTED
- GET_INSTANCE_INVALID_PARAMETER,
- GET_INSTANCE_ACCESS_DENIED,
- GET_INSTANCE_FAILED,

Returning any other integer value causes a compilation error, since the return type is a C++ enumeration.

Performance. At some point for providers that generate large numbers of instances (and/or associations and references) the performance of using an implementation that only implements the enumerate_instances function can have limitations. Thus, it is illogical to generate thousands of instances if only one is required. It may be illogical to generate thousands of instances for an association if the filters limit the output to a very small subset. However, we would propose that it is probably best to experiment with the defaults before implementing the optional functions. Most providers are only involved in a limited number of instances and the reduction in programmer time for a result that is always correctly generated without duplicate functionality (ex. generating instances in both the getInstance and enumerateInstance functions) is a strong argument for developer efficiency and even provider efficiency.

3.7 Building the Provider

Building the provider involves making a shared library (or DLL) out of source files created in this chapter, which includes:

```
President.cpp
repository.cpp
President_Provider.cpp
module.cpp
```

The subsections below discuss general issues associated with building a provider. This section does *not* explain how to compile C++ sources, how to build shared libraries, nor how to write makefiles. These activities are particular to your environment and are beyond the scope of CIMPLE.

3.7.1 Enabling a Provider Entry Point

The `module.cpp` file conditionally defines entry points for every supported provider interface (CMPI, OpenPegasus, OpenWBEM, and WMI). To enable compilation of an entry point, define one of the following macros while compiling `module.cpp`.

- `CIMPLE_CMPI_MODULE` (for CMPI)
- `CIMPLE_PEGASUS_MODULE` (for OpenPegasus)
- `CIMPLE_OPENWBEM_MODULE` (for OpenWBEM)
- `CIMPLE_WMI_MODULE` (for WMI)

For example, to enable the CMPI entry point, pass the following option to the compiler when compiling `module.cpp`.

```
    -DCIMPLE_CMPI_MODULE
```

3.7.2 Locating the CIMPLE Include Directory

Be sure the compiler can locate the CIMPLE include directory. If CIMPLE was installed in a standard system location, you may not need to do anything. Otherwise, pass the include path as a compiler option. For example, if CIMPLE was installed under `/xyz`, then pass the following option to the compiler.

```
    -I/xyz/include
```

3.7.3 Linking the CIMPLE Libraries

The library (or DLL) must be linked with the correct CIMPLE libraries, which includes `cimple`,one of the following adapter libraries and interface-specific libraries as defined in section 3.7.4.

- `cimplecmpiadap` (for !adapter library
- `cimplepegadap` (for !adapter library)

- `cimpleowadap` (for !adapter library)
- `cimplewmiadap` (for !adapter library)

Static linking. CIMPLE 1.0.0 supports of these libraries. To link statically, CIMPLE must be configured with the `--enable-static` option.

3.7.4 Linking the Interface-Specific Libraries

The library (or DLL) must be linked with libraries required by the specific provider interface. These are identified in table 3.1. Note that CMPI requires no libraries.

Table 3.1: **Interface-Specific Libraries**

Provider Interface	**Required Libraries**
CMPI	
!Interface Specific Libraries	`pegprovider, pegcommon`
!Interface Specific Libraries	`owprovider, owcppprovifc, openwbem`
!Interface Specific Libraries	`ole32.lib` and `oleaut32.lib`

3.7.5 Position-Independent Code

On !compile options systems, the sources must be compiled with the `-fPIC` option (to generate position-independent code suitable for shared libaries). If you forget, your provider may fail to load.

3.8 Registering the Provider

Today provider registration is specific to each CIM Server including the variety of CIMServers supported by CIMPLE. The complete definition of provider registration for each of the supported CIM Servers is detailed in section 15.

This example is specific to provider registration because Pegasus is a simple example and CIMPLE provides an tool for automating Pegasus provider registration. Pegasus requires formal registration of providers via a set of Pegasus specific provider registration classes. This involves creating registration instances in the OpenPegasus repository. Within the native Pegasus environment you do this by manually defining a MOF file containing the registration instances and compiling it with the OpenPegasus `cimmof` command. Appendix D contains the MOF file you would have to write to register our `President` provider.

The CIMPLE `regmod` tool automates this registration process. To register the `President` provider, first be certain the Pegasus server is running, and then type this command:

```
$ regmod -c libPresident.so
Using CMPI provider interface
Registering President_Provider (class President)
```

The `regmod -c` option creates any classes the provider module uses that are not already in the Pegasus repository. For our provider, it creates the `President` class the first time it runs.

Sometimes you may need the `regmod -d` option that dumps the MOF registration instances required to register the provider, without actually registering anything or modifying the Pegasus repository. For more on the `regmod` tool type:

```
regmod -h
```

This is much easier than the manual approach, which involves writing MOF registration instances and compiling them with the OpenPegasus `cimmof` tool.

3.9 Testing the Provider

Again testing may be CIM Server specific because the client test tools and client APIs are server specific.

3.9.1 Testing With Pegasus

Finally, we are ready to test our provider with the OpenPegasus server. The following subsections describe the steps. We assume you have already registered the provider, as described in section 3.8.

We are using as an example testing with Pegasus. Take take a moment to locate the OpenPegasus `CLI` tool, called `cimcli` in OpenPegasus Version 2.7.0 and later. We use the name `cimcli` in the examples below. If you are using another environment where `cimcli` is not available you may find an equivalent client tool that can generate CIM Operations, receive operation responses, and invoke CIM methods.

You should use the `configure` option `--DEBUG` when testing. There are a number of tests that are added to the `CIMPLE` environment when `DEGUG` is set to catch errors (ex. String index limits) which are removed in when the DEBUG option is not used. Additionally, the log functions 10 may be used to generate diagnostic output.

3.9.2 Testing With SFCB

With SFCB either the specific SFCB clients or the Pegasus `cimcli` client can be used to generate CIM Operations.

3.9.3 Testing With WMI

None of the tools from other CIM Servers work with WMI servers. The interfaces to WMI are completely different. For simple tests it is easiest to use one of the provided by (`wbemtest`, or `WBEM Studio`, or `wmic` as a test tool to generate operations and view the instances/ classes available from the WMI server. `wbemtest` is normally available on the windows system and Microsoft `WBEM Studio` or `wmic` can be downloaded from !test tools.

We have included some `wmic` examples with the wmi test providers in the CIMPLE release.

3.9.4 Installing the Provider

As with registration, installation is CIM Server specific.

NOTE: The installation described here is specific to the Pegasus platform.

Copy `libPresident.so` to the OpenPegasus provider directory, where OpenPegasus finds its provider libraries. If you are using the OpenPegasus source distribution, the provider directory is here:

```
$PEGASUS_HOME/lib
```

If you are using the OpenPegasus RPM, then the location is installation dependent.

3.9.5 Enumerating Instance Names Test

To enumerate instance names `President`, type the following command.

```
$ cimcli ni President
President.Number=1
President.Number=2
President.Number=3
```

3.9.6 Enumerating Instances Test

To enumerate instances of `President`, type the following command.

```
$ cimcli ei President
instance of President
{
    Number = 1;
    First = "George";
    Last = "Washington";
};
instance of President
{
    Number = 2;
    First = "John";
    Last = "Adams";
};
instance of President
{
    Number = 3;
    First = "Thomas";
    Last = "Jefferson";
};
```

3.9.7 Getting an Instance Test

To get an instance of `President`, type the following command.

```
$ cimcli gi President.Number=1
instance of President
{
    Number = 1;
    First = "George";
    Last = "Washington";
};
```

Chapter 4

CIMPLE Data Types

This chapter describes the CIMPLE data types, used to represent the . Table **??** shows the correspondence between the two.

Table 4.1: **CIM Data Types and CIMPLE Mapping**

CIM Data Type	CIMPLE Data Type
boolean	`boolean`
uint8	`uint8`
sint8	`sint8`
uint16	`uint16`
sint16	`sint16`
uint32	`uint32`
sint32	`sint32`
uint64	`uint64`
sint64	`sint64`
real32	`real32`
real64	`real64`
char16	`char16`
string	`String`
datetime	`Datetime`

All `CIMPLE` data types are defined in the `cimple` namespace . are formed with the `Array` class, discussed in section 4.7. These data types are discussed in the following sections.

4.1 Booleans

CIM booleans are represented with the CIMPLE `boolean` type, which is merely a type definition of the C++ `bool` type.

```
boolean test;
boolean test2 = test;
test = true;
test2 = false;
if(test)
    ....;
```

4.2 Integers

IndexCIM integers are represented by CIMPLE data types with the same name. Table 4.2 shows the corresponence between the CIMPLE data types and C++ types.

Table 4.2: **Integer Data Types and C++ Mapping**

CIMPLE Type Name	**C++ Type**
`uint8`	`unsigned char`
`sint8`	`signed char`
`uint16`	`unsigned int`
`sint16`	`signed int`
`uint32`	`unsigned long`
`sint32`	`signed long`
`uint64`	`unsigned long long (GCC)` `unsigned __int64 (MSVC)`
`sint64`	`signed long long (GCC)` `signed __int64 (MSVC)`

We recommend using the CIMPLE type name to promote portability.

4.3 Reals

CIM reals are represented by CIMPLE data types with the same name. Table 4.3 shows the correspondence between the CIMPLE data types and C++ types.

Table 4.3: **Real Data Types**

CIMPLE Type Name	C++ Type
`real32`	`float`
`real64`	`double`

Although the CIMPLE data type and the corresponding C++ type are interchangeable, we recommend using the CIMPLE type name for clarity.

4.4 Char16

The `char16` class implements the CIM **char16** type. This class encapsulates a `uint16` character code, which is zero be default. `Char16`'s can be constructed and initialized from `uint16`'s and other `char16`'s. The following source snippet illustrates some of the typical operations.

```
char16 w = 65;
char16 x = w;
char16 y;
y = 66;
char16 z;
z = y;

printf("%u %u %u %u\n", w.code(), x.code(), y.code(), z.code());
```

4.5 Strings

CIMPLE provides a `String` class for representing CIM strings. This class defines the essential operations for building and manipulating sequences of 8-bit characters. A `String` can contain UTF-8 strings, although there are no special operations for processing them as such. For example:

- `String::size` returns the number of bytes in a string, not the number of characters.
- `String::operator[]` returns the i-th byte in the string, not i-th character.

The functions available for String manipulation include:

- `constructor` create a new String.
- `assign` assign one string to another.
- `assign` assign one string to another.
- `append` append to a String.
- `clear` remove the data from a String.
- `reserve` reserve memory for a String.
- `substr` remove a substring from a String.
- `find` find a character or substring in a String.
- `equal` test a String for equality with another String.
- `equal` test a String for equality with another String.
- `size` determine the number of bytes in a string.
- `c\_str` access the C string data in a String.
- `[]` Access one element in a String.

Note that most of the functions that take a String as parameter also take char * as input value.
We suggest obtaining an internationalization/localization package if you need to process the contents of a UTF-8 string.

The following example illustrates a few of the essential string operations.

```
String dow = "Red Green Blue";

String red = dow.substr(0, 3);
dow.remove(0, 4);
dow.append(" Yellow");
const char* str = dow.c_str();
```

4.6 Datetime

The `Datetime|` class implements the CIM **datetime** type. A datetime represents either a *timestamp* or an *interval*. A timestamp has the following string format, whose fields are defined in table 4.4.

```
yyyymmddhhmmss.mmmmmmsutc
```

Table 4.4: **Timestamp Fields**

Field	Meaning
`yyyy`	year
`mm`	month
`dd`	day
`hh`	hour
`mm`	minutes
`ss`	seconds
`mmmmmm`	microseconds
`s`	sign ('+' or '-')
`utc`	UTC offset

An interval has the following string format, whose fields are defined in table 4.5.

```
ddddddddhhmmss.mmmmmm:000
```

Table 4.5: **Interval Fields**

Field	Meaning
`dddddddd`	days
`hh`	hours
`mm`	minutes
`ss`	seconds
`mmmmmm`	microseconds
`:`	signifies an interval
`000`	always '000' for intervals

The `Datetime` class is constructed from either string format. For example:

```
    Datetime timestamp("20060101120000.000000+360");
    Datetime interval("00000100010203.000000:000");
```

The `Datetime::ascii` method converts the datetime back to string format. For example, the following snippet gets and prints the timestamp constructed above.

```
    String str = timestamp.ascii();
    printf("timestamp=%s\n", str.c_str());
```

We can "prettify" the string format by passing `true` to `Datetime::ascii` as shown in the following code fragment.

```
    String str = timestamp.ascii(true);
    printf("timestamp=%s\n", str.c_str());
```

This produces a slighty more readable string format:

```
2006/01/01 12:00:00.000000+360
```

4.7 Arrays

The CIMPLE `Array` class is used to form arrays of any of the CIM data types discussed in this chapter.

The `Array` class includes the following functions:

- `Constructor()` construct an array of any of the CIM Data types.
- `append()` add an element to the end of the array.
- `insert()` add an element into the array at the position defined by the index parameter.
- `prepend()` add an element at the beginning of the array.
- `remove()` add one or more elements from the array.
- `clear()` remove all the elements from the array.
- `size()` determine the number of elements in an array.
- `reserve()` prereserve memory for the array (optimization).
- `String::operator[]()` returns the i-th element in the array.

This class is similar to the STL `vector` class but offers three major advantages:

- It causes virtually no code bloat due to template usage.
- It offers binary compatibility from one release to the next.
- It provides a simplified index-oriented interface (no iterators).

Although `Array` is a template class, it does NOT cause object code bloat the way most template classes do. Each template member function is a trivial one-line wrapper that calls a common non-template function (examine the implementation if you are curious).

4.7.1 Array Construction

The `Array` template class can construct arrays of any of the CIM Data types:

```
Array<uint32> a;
Array<String> s;
Array<DateTime> dt;
```

4.7.2 Inserting Elements into an array

For example, the following builds and prints an array of strings.

```
Array<String> a;
a.append("Red");
a.append("Green");
a.append("Blue");

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

There are also forms of the append, prepend and insert functions functions for adding more than one element as shown below:

```
Array<int> a;
int elements[] = { 10, 20, 30, 40, 50 };
a.append(elements, 5);
```

4.7.3 Removing Elements from an Array

The `Array` class provides two methods for removing elements from an array: one that removes a single element and one that removes one or more elements. The following snippet illustrates both:

```
Array<int> a;
a.append(0);
a.append(1);
a.append(2);
a.append(3);
a.append(4);

// Remove first element:
a.remove(0);

// Remove last element:
a.remove(a.size()-1);

// Remove last two elements.
a.remove(1, 2);
```

4.7.4 Array size

4.7.5 Clearing an Array

The `clear()()` function removes all elements from an array. After calling `clear()`, the `size()` function will return zero. But clearing an array does not necessarily reclaim any memory used by the array. The memory is retained to reduce overhead associated with any future insertions. To reclaim the memory, the array itself must be destructed.

4.7.6 Reserving Memory for an Array

The `Array` class automatically expands the internal memory allocation as elements are added. Extra space may be reserved for future additions to limit resource usage. For example, an array with five elements may have space for as many as eight. In general, allocations are rounded up to the next power of two. Like STL vectors, the

`Array` class provides a `reserve()` function that makes space for so many elements. But note that `reserve()` does not change the size of the array. If you know you are about to create an array with 100 elements, it will reduce allocation and copying by reserving the memory up front. The following snippet will incur at most one allocation.

```
Array<uint32> a;
a.reserve(100);

for (size_t i = 0; i < 100; i++)
    a.append(i);
```

4.7.7 Array Class Error Checking

The `Array` class (like the functions in the standard C library) performs no error checking on input arguments. For example, passing a null pointer to a member function will result in a crash. Similary, passing an out-of-range index will have unpredictable results. It is the caller's responsibility to avoid these errors. Explict error checking would make the implementation bigger and slower, which is mainly why the C library routines do not check for errors either. For example, the following use of strcpy() will surely cause a crash.

```
// core dump!
strcpy(NULL, NULL);
```

4.7.8 Octets Strings

CIMPLE includes two functions used to translate between as received defined in the CIM Infrastructure specification with the OctetString Qualifier and an internal form useful for direct access to the data in the OctetString. Today these functions are used only to convert between the array of strings form for Octet strings, not the alternate form (array of uint8). Note that we handle only single strings. The mapping of the array itself is left to the user.

The functions are:

- octets_to_string
- string_to_octet

octets to strings

This function converts an octet string to a String object. The `data` parameter is a pointer to an octet string with `size` octets. This function returns a string that conforms to the following grammar:

```
        "0x"4*(<hexDigit><hexDigit>)

    For example, the following octet string:

        01 02 03 04

    is encoded as follows ("0x00000008" followed by "01020304").

        0x0000000801020304

    The leading "0x00000008" gives the total number of hex digits in the
    encoding.

    The empty octet string is encoded as follows.

        0x00000004

    Note that there are a total 4 hex digits in this encoding.
```

The definition of the function is as follows:

```
void String octets_to_string(
    const unsigned char* data,
    uint32 size);
```

The following is an example of this function:

```
        unsigned char data1[] = { 1, 2, 3, 4, 5, 6 };
        String s = octets_to_string(data1, sizeof(data1));
        assert(s == "0x0000000A010203040506");
```

string to octet

This function converts a String object to an octet string. Note that the string must conform to the standard encoding described in METHODoctets_to_string. The `str` parameter contains the text string to be converted. Up to `size` octets are written to the `data` parameter.

This function returns -1 if the string encoding is invalid. Otherwise it returns the size of the encoded string. A return value larger than `size` indicates that `data` was too small to receive the resulting octet string. In that case, `string_to_octets()` will succeed if called again with a `size` parameter that is at least as large as the initial return value.

```
    void ssize_t string_to_octets(
        const String& str,
        unsigned char* data,
        uint32 size);
```

This function is a simpler form of `string\_to\_octets()`, which leaves the resulting octets in an array of unsigned chars. It returns -1 if the the string encoding is invalid. Otherwise it returns 0. We recommend that the user use this rather than `octets_to_string()` above because of its simpler interface.

```
int string_to_octets(
    const String& str,
    Array<unsigned char>& octets);
```

The following is an example of each of these functions:

```
        unsigned char data1[] = { 1, 2, 3, 4, 5, 6 };
        String s = octets_to_string(data1, sizeof(data1));
        assert(s == "0x0000000A010203040506");
        assert(string_to_octets(s, 0, 0) == 6);
        unsigned char data2[] = { 0, 0, 0, 0, 0, 0 };
        ssize_t n = string_to_octets(s, data2, sizeof(data2));
        assert(n == 6);
        assert(memcmp(data1, data2, sizeof(data1)) == 0);
```

Chapter 5

Working With CIM Instances

This chapter shows how to use CIM classes generated by the `genclass` tool. All examples in this chapter are based on the following MOF class (`Employee` class, `Manager` Class, `Link` Association Class) definitions, contained in the `repository.mof` file for the provider.

```
class Employee
{
    [Key] uint32 Id;
    string First;
    string Last;
    [Values{"Male", "Female"}, ValueMap{"1", "2"}]
    uint32 Gender;
    boolean Active = true;
    boolean OutOfOffice;
};
```

```
class Manager : Employee
{
    uint32 NumEmployees;
    uint32 Budget;
};
```

```
[Association]
class Link
{
    [Key] Employee REF Emp;
    [Key] Manager REF Mgr;
};
```

The following sections discuss various issues associated with using instances.

5.1 Generating the Classes

The following command (`genclass`) generates C++ clases from the MOF class definitions shown above.

```
$ genclass -r Employee Manager Link
Created Employee.h
Created Employee.cpp
Created Manager.h
Created Manager.cpp
Created Link.h
Created Link.cpp
created repository.h
Created repository.cpp
```

`genclass` reads `repository.mof` from the current directory. Here is the resulting `Manager` class.

```
class Manager : public Instance
{
public:
    // Employee features:
    Property<uint32> Id;
    Property<String> First;
    Property<String> Last;
    Property<uint32> Gender;
    Property<boolean> Active;
    Property<boolean> OutOfOffice;

    // Manager features:
    Property<uint32> NumEmployees;
    Property<uint32> Budget;

    CIMPLE_CLASS(Manager)
};
```

Notice this class explicity defines inherited properties before defining its own properties. By "flattenning out" classes in this way, CIMPLE supports static and dynamic casting, described in section 5.7.

5.2 The Property Structure

All generated class properties are represented by the `Property` template structure, defined as follows.

```
template<class T>
struct Property
{
    T value;
    uint8 null;
    void set(const T& x);
    void clear();
};
```

The `value` field contains the property value; whereas the `null` field indicates whether the property is null. The following code fragment sets a property value and clears its null flag.

```
Property<uint32> x;
x.value = 99;
x.null = false;
```

This is equivalent to calling the `set` member function as follows.

```
Property<uint32> x;
x.value.set(99);
```

To clear the value and set the null flag, do this indexProperty!set null.

```
Property<uint32> x;
x.value = 0;
x.null = true;
```

This is equivalent to calling the `clear` member function as follows .

```
Property<uint32> x;
x.clear();
```

We recommend using the `set` and `clear` functions exclusively rather than modifying the fields directly. Forgetting to set or clear a field is easy and using these functions will prevent this. Table 5.1 summarizes these two functions.

Table 5.1: **Set and Clear**

Function	**Description**
`Property<T>::set(const T& value)`	set value; clear null flag
`Property<T>::clear()`	clear value; set null flag

In section 5.6, we show how to use properties as members of generated classes. In fact, the `Property` structure is never used apart from generated classes. We only do so here to illustrate their usage.

5.3 Instance Lifecycle Operations

This section shows how to create, clone, and destroy instances. For every class compiled, `genclass` generates the following three member functions.

```
create
clone
destroy
```

The subsections below discuss the role of these methods.

5.3.1 Creating an Instance

Every generated class defines a static `create` member function. The following code fragment uses this method to create an instance of `Manager`.

```
Manager* m = Manager::create();
print(m);
```

To examine the new instance, we call the `print` method as follows.

```
print(m);
```

This prints the following to standard output (we discuss the `__name_space` member in section 5.10).

```
Manager
{
    string __name_space = "";
    uint32 Id = 0;
    string First = "";
    string Last = "";
    uint32 Gender = 0;
    boolean Active = false;
    boolean OutOfOffice = false;
    uint32 NumEmployees = 0;
    uint32 Budget = 0;
}
```

Calling `create` with no arguments (or with `false`), creates an "uninitialized" instance, with non-null properties whose values are empty. Table 5.2 shows the empty values for the various data types.

Table 5.2: **Empty Values**

Data Type	**Empty Value**
booleans	false
integers	zero
reals	zero
char16	zero
string	empty string
datetime	zero interval
array	empty array

Alternatively, you can create an "initialized" instance by passing `true` as an argument to `create` as follows.

```
Manager* m = Manager::create(true);
print(m);
```

This creates an instance whose property values are initialized according to the MOF class definition. If a class property has an initializer, the instance property receives the same value; otherwise, the property is set to null. Printing this instance produces:

```
Manager
{
    string __name_space = "";
    uint32 Id = NULL;
    string First = NULL;
    string Last = NULL;
    uint32 Gender = NULL;
    boolean Active = true;
    boolean OutOfOffice = NULL;
    uint32 NumEmployees = NULL;
    uint32 Budget = NULL;
}
```

The `Active` property is `true` since that property in the MOF class definition has an explicit initializer with that value. All other properties are null, since the MOF class definition specifies no value for those properties.

> **Operator `new`**. The C++ `new` operator does not work on CIMPLE instances. Use the `create` method instead.

5.3.2 Cloning an Instance

Every generated class defines a `clone` member function. The following code fragment uses this method to clone a `Manager` instance.

```
Manager* m1 = Manager::create(true);
Manager* m2 = m1->clone();
```

The cloned instance is identical to the original instance in every respect.

5.3.3 Destroying an Instance

Every generated class defines a static `destroy` member function. The following code fragment uses this method to destroy a `Manager` instance.

```
Manager* m = Manager::create(true);
Manager::destroy(m1);
```

Alternatively, you can call `destroy` as shown below.

```
destroy(m1);
```

Destroying an instance reclaims all heap memory associated with that instance.

> **Operator `delete`**. The C++ `delete` operator does not work on CIMPLE instances. Use the `destroy` method instead.

5.4 Reference Counting

Generated classes support thread-safe reference counting. Instances are created with an initial reference count of 1. The `ref` and `unref` functions respectively increment and decrement the reference count. `Unref` destroys an instance when the reference count becomes zero. The following example illustrates the use of reference counts.

```
// Create instance with a reference count of 1.
Manager* m = Manager::create(true);

// Increase reference count to 2.
ref(m);

// Decrease reference count to 1.
unref(m);

// Decrease reference count to 0 and destroy instance.
unref(m);
```

The effect of reference counting on various functions is summarized in table 5.3.

Table 5.3: **Reference Counting Notes**

Function	**Notes**
`create`	Initializes reference count to 1.
`clone`	Initializes reference count to 1.
`ref`	Increments reference count.
`unref`	Decrements reference count and destroys instance if zero.
`destroy`	Assert on debug builds if reference count is not 1.

The `Ref` class is a “smart pointer” that uses reference counting to manage the lifetime of instances. For example, the following fragment uses the `Ref` class to manage a `Manager` instance.

```
Ref<Manager> m = Manager::create(true);
m->Id.set(1);
m->First.set("Jane");
m->Last.set("Do");
m->Gender.set(2);
m->Active.set(true);
m->NumEmployees.set(10);
m->Budget.set(1000000);
print(m.ptr());
```

The `Manager` instance is automatically released when `m` destructs. Table 5.4 summarizes key member functions of the `Ref` class.

Table 5.4: **Ref Member Functions**

Member Function	Description
`reset()`	unreference instance; set pointer to zero
`reset(T* ptr)`	unreference instance; set pointer to ptr argument
`ptr()`	return pointer to instance
`steal()`	return pointer to instance; set pointer to zero
`count()`	return current reference count

5.5 References

CIMPLE has no reference type *per se*. Instead references—sometimes called object paths—are represented by ordinary instances of the given class. For example, take the following object path.

```
Manager.Id=1
```

We represent this with the following code fragment.

```
Manager* m = Manager::create(true);
m->Id.set(1);
```

When used as a reference, the non-key properties are ignored. The following prints just the key fields of the `Manager` instance created above.

```
print(m, true);
```

This produces the following output.

```
Manager
{
    string __name_space = "";
    uint32 Id = 1;
}
```

References are used to define association end-points. The following illustrates how to create an association, of class `Link`, between an `Employee` and a `Manager`.

```
Employee* e = Employee::create(true);
e->Id.set(1);

Manager* m = Manager::create(true);
e->Id.set(2);

Link* link = Link::create(true);
link->Emp = e;
link->Mgr = m;

print(link);
```

The `print` function prints the following.

```
Link
{
    string __name_space = "";
    Manager Mgr =
    {
        string __name_space = "";
        uint32 Id = 2;
    }
    Employee Emp =
    {
        string __name_space = "";
        uint32 Id = 1;
    }
}
```

See chapter 8 for more on creating association instances.

5.6 Working With Properties

In this section we show how to use instance properties. We continue with the `Manager` class example (defined on page 44 and generated on 45). The code fragment below creates, initializes, and prints an instance of this class.

```
Manager* m = Manager::create(true);
m->Id.set(1);
m->First.set("Jane");
m->Last.set("Do");
m->Gender.set(2);
m->NumEmployees.set(10);
m->Budget.set(1000000);
print(m);
```

The `print` function produces the following output.

```
Manager
{
    string __name_space = "";
    uint32 Id = 1;
    string First = "Jane";
    string Last = "Do";
    uint32 Gender = 2;
    boolean Active = true;
    boolean OutOfOffice = false;
    uint32 NumEmployees = 10;
    uint32 Budget = 1000000;
}
```

Notice that we did not explicitly set the `Active` field. Instead we accepted the default value of `false` specified by the MOF class definition.

Recall from section 5.2, that the `clear` member function clears a property's value and sets its null flag. For example, the following fragment clears the `Budget` property.

```
m->Budget.clear();
print(m);
```

The abbreviated output is shown below.

```
Manager
{
    ...
    uint32 Budget = NULL;
    ...
}
```

Getting a property's `value` and `null` fields is straightforward. The following checks whether the given manager has a budget, and if so prints out that budget.

```
if (!m->Budget.null)
    printf("Budget: %u\n", m->Budget.value);
```

You can directly modify the `value` and `null` fields, but we recommend using the `set` and `clear` functions instead to avoid errors.

5.7 Casting

This section explains how casting works in CIMPLE. We discuss the , *static casting*, and *dynamic casting*. We preface our discussion with a cautionary note. Never use the C++ `dynamic_cast` operator on CIMPLE instances. Generated classes are non-virtual, so the `dynamic_cast` operator does not apply to them. Section 5.7.3 discusses the alternative to `dynamic_cast`.

5.7.1 The CIMPLE Inheritance Model

You might have noticed that all generated classes above derive from `Instance`. You might wonder then how inheritance works and why we did not use ordinary C++ inheritance. This subsection answers both questions.

To illustrate how inheritance works, we consider the following MOF definitions.

```
class A
{
    [Key] uint32 w;
};
```

```
class B : A
{
    boolean x;
    string y;
};
```

```
class C : B
{
    datetime z;
};
```

These definitions define three classes: `A`, `B`, and `C`. `C` is a subclass of `B`, which is a subclass of `A`. Now we examine the generated C++ classes.

```
class A : public Instance
{
public:
    // A features:
    Property<uint32> w;

    CIMPLE_CLASS(A)
};
```

```
class B : public Instance
{
public:
    // A features:
    Property<uint32> w;

    // B features:
    Property<boolean> x;
    Property<String> y;

    CIMPLE_CLASS(B)
};
```

```
class C : public Instance
{
public:
    // A features:
    Property<uint32> w;

    // B features:
    Property<boolean> x;
    Property<String> y;

    // C features:
    Property<Datetime> z;

    CIMPLE_CLASS(C)
};
```

Each class defines inherited members first followed by its own members. For example, B defines the property inherited from A before its own properties. Similarly, C defines the properties inherited from A and B before its own properties. So the initial segment of any class has the same layout as its superclass, which means that any instance can be substituted for an instance of the superclass (through casting).

You might wonder why CIMPLE inheritance is not implemented using ordinary C++ inheritance. Unfortunately, C++ does not permit a derived class to change the type of a data member, which is required in CIM. For example, the following MOF definition changes the class of a reference.

```
[Association]
class AA
{
    [Key] A ref left;
    [Key] A ref right;
};

[Association]
class BB : AA
{
    [Key] B ref left;
    [Key] B ref right;
};
```

In this example, BB changes the class of the inherited `left` and `right` references, from A to B.

5.7.2 Static Casting

As mentioned above, the initial segment of any class has the same layout as the superclass. This characteristic makes it possible to treat an instance of a class as an instance an ancestor class. The following code fragment casts an instance from class C to class B.

```
C* c = C::create(true);
B* b = reinterpret_cast<B*>(c);
```

Similary, the following fragment casts an instance from class C to class A.

```
C* c = C::create(true);
A* a = reinterpret_cast<A*>(c);
```

In both examples we use the C++ `reinterpret_cast` operator to perform the cast. This operator can be dangerous since it circumvents the type system. When you use this operator, be certain that the source class is in fact an instance of the target class.

5.7.3 Dynamic Casting

As mentioned already, the C++ `dynamic_cast` operator does not work on CIMPLE classes, which are are non-virtual and do not employ conventional inheritance. Alternatively, CIMPLE provides the `cast` operator. The following fragment illustrates *down-casting* (i.e., casting from an ancestor class to a descendent class).

```
void f(A* a)
{
    C* c = cast<C*>(a);

    if (c)
    {
        // a is an instance of C.
    }
}
```

The cast returns a non-zero pointer if `a` refers to an instance of class `C` or an instance derived from class `C`.

Alternatively, we can cast in the other direction. The following illustrates *up-casting* (i.e., casting from a descendent class to an ancestor class).

```
void f(C* c)
{
    A* a = cast<A*>(c);

    if (a)
    {
        // c is an instance of A.
    }
}
```

The cast returns a non-zero pointer if `c` refers to an instance derived from class `A`. Unlike C++, in which up-casting is implicit, CIMPLE up-casting requires an explict cast.

5.8 Embedded Objects

This section explains how CIMPLE represents CIM embedded objects. Recall that in the CIM model today a string property bearing the `EmbeddedObject` qualifier may contain a class or an instance. The following MOF definition defines a class with a single embedded object property.

```
[Indication]
class OutOfOfficeNotice
{
    [EmbeddedObject]
    string employee;
};
```

Fortunately, CIMPLE does not require developers to encode the embedded object as a string. Instead, CIMPLE generates the following class, containing an instance pointer rather than a string property.

```
class OutOfOfficeNotice : public Instance
{
public:
    // OutOfOfficeNotice features:
    Instance* employee;

    CIMPLE_CLASS(OutOfOfficeNotice)
};
```

The following code fragment creates an instance of `OutOfOfficeNotice`, whose `employee` property refers to an instance of `Employee`.

```
Employee* e = Employee::create(true);
e->Id.set(1001);

OutOfOfficeNotice* o = OutOfOfficeNotice::create();
o->employee = e;
print(o);
```

This fragment produces the following output.

```
OutOfOfficeNotice
{
    string __name_space = "";
    Employee employee =
    {
        string __name_space = "";
        uint32 Id = 1001;
    }
}
```

Recall that an embedded object can refer to either a class or an instance. Classes are represented by an instance with null key values. Instances are represented by an instance with non-null key values. A null embedded object pointer is an error.

5.9 Embedded Instances

Whereas embedded objects are defines as part of indication support, embedded instances are defined through the embedded_instance qualifier as part of extrinsic methods. A method must be capable of receiving an embedded_instance as an IN parameter or generating one as an OUT parameter.

CIMPLE 1.0.22 and subsequent versions support embedded instances. Table 5.5 shows the status of embedded instances in recent versions of CIMPLE, Pegasus, and CIM. Note that embedded instance support is an optional configuration function (when CIMPLE is configured with –enable-embedded-instances) largely because the implementation of embedded instances is inconsistent for the difference CIM Servers.

Pegasus 2.6.0 and subsequent versions support embedded instances. The DMTF CIM 2.15 and subsequent releases support embedded instances.

Table 5.5: **Embedded Instance Support**

Version	Supported
CIMPLE 0.99.56	no
CIMPLE 0.99.40	no
CIMPLE 0.99.34	no
CIMPLE 1.0.0	no
CIMPLE 1.0.22 and subsequent	yes
Pegasus 2.5.2	no
Pegasus 2.5.3	no
Pegasus 2.5.4	no
Pegasus 2.6.0 and subsequent	yes
CIM 2.11	no
CIM 2.12	no
CIM 2.13.1	no
CIM 2.13.1 experimental	yes
CIM 2.14	no
CIM 2.14 experimental	yes
CIM 2.15 and subsequent	yes

5.9.1 Implementing Embedded Instances

We have an example of embedded instance definition which includes an embedded instance both as a property in the class and in the defined method:

```
// First embedded Class
class CMPL_Embedded1
{
    [Key] uint32 key;
};

// Second Embedded Class
class CMPL_Embedded2
{
    [Key] uint32 key;
};

// Class for which Provider is defined. Contains embedded
// instance property and Embedded Instance parameter

class CMPL_Embedded
{
    [Key]
    uint32 Key;

    // define an embedded instance as a property

    [EmbeddedInstance("CMPL_Embedded1")]
    string embedded1;

    string foo(
        // Return an embedded instance as a return property of a method.

        [EmbeddedInstance("CMPL_Embedded2"), In(false), Out(true)]
        string arg1);
};
```

Compiling this mof with `genclass` and `genprov` as follows.

```
genclass -r CMPL_Embedded
genprov CMPL_Embedded
genmod CMPL_Embedded CMPL_Embedded
```

Note that we are generating for CMPL_Embedded and that the are automatically generated.

This generates the following files:

```
Created CMPL_Embedded1.h
Created CMPL_Embedded1.cpp
Created CMPL_Embedded2.h
Created CMPL_Embedded2.cpp
Created CMPL_Embedded.h
Created CMPL_Embedded.cpp
created repository.h
Created repository.cpp
```

We can implement the provider very simply to include the embedded instance using the standard definition of create to define the embedded instance as follows:

```
Enum_Instances_Status CMPL_Embedded_Provider::enum_instances(
    const CMPL_Embedded* model,
    Enum_Instances_Handler<CMPL_Embedded>* handler)
{
    // Create the instance
    CMPL_Embedded* e = model->clone();

    // set the key property in place
    e->Key.set(12345);

    // Create the embedded instance and place into the enumerted instance
    {
        CMPL_Embedded1* e1 = CMPL_Embedded1::create(true);
        e1->key.set(9999);
        e->embedded1 = e1;
    }

    handler->handle(e);

    return ENUM_INSTANCES_OK;
}
```

/noindent To implement the embedded instance parameter in the extrinsic method:

```
Invoke_Method_Status CMPL_Embedded_Provider::foo(
    const CMPL_Embedded* self,
    const CMPL_Embedded1* arg1,
    Property<String>& return_value)
{
    // Create the embedded instance for arg1
    {
        CMPL_Embedded1* e1 = CMPL_Embedded2::create(true);
        e1->key.set(11111);
        arg1 = e1;
    }

    return_value.set("Hello");

    return INVOKE_METHOD_OK;
}
```

5.10 The `__name_space` member

Every generated class has a `__name_space` member. Association providers use this member to build cross-namespace association providers. For example, the following fragment creates a cross-namespace association instance.

```
Employee* e = Employee::create(true);
e->__name_space = "root/abc";
e->Id.set(1);

Manager* m = Manager::create(true);
m->__name_space = "root/xyz";
e->Id.set(2);

Link* link = Link::create(true);
link->Emp = e;
link->Mgr = m;

print(link);
```

This example is identical to the one presented in section 5.5, except for two additional lines that set the `__name_space` member. This fragment produces the following output.

```
Link
{
    string __name_space = "";
    Manager Mgr =
    {
        string __name_space = "root/xyz";
        uint32 Id = 2;
    }
    Employee Emp =
    {
        string __name_space = "root/abc";
        uint32 Id = 1;
    }
}
```

The `__name_space` member is rarely used outside of cross-namespace associations. When omitted, it defaults to the originating namespace of the request.

Chapter 6

Instance Providers

This chapter shows how to develop a complete instance provider, which supports all instance provider methods, shown in the following table.

Table 6.1: **CIMPLE Instance Provider Methods**

Instance Provider Methods
FNCT[Instance Provider Method]load
FNCT[Instance Provider Method]unload
FNCT[Instance Provider Method]get_instance
FNCT[Instance Provider Method]enum_instances
FNCT[Instance Provider Method]create_instances
FNCT[Instance Provider Method]delete_instances
FNCT[Instance Provider Method]modify_instances

Our provider implements the `Employee` class, introduced in chapter 5.

```
class Employee
{
    [Key] uint32 Id;
    string First;
    string Last;
    [Values{"Male", "Female"}, ValueMap{"1", "2"}]
    uint32 Gender;
    boolean Active = true;
    boolean OutOfOffice = false;
};
```

This time we use the `genproj` tool rather than running `genclass`, `genprov`, and `genmod` independently. The following command generates all the sources required by our `Employee` provider.

```
$ genproj Employee Employee
==== genclass:
Created Employee.h
Created Employee.cpp
created repository.h
Created repository.cpp
==== genprov:
Created Employee_Provider.h
Created Employee_Provider.cpp
==== genmod:
Created module.cpp
```

The complete source for this provider is included in the CIMPLE source release under:

```
cimple-1.0.0/src/provider/Employee
```

6.1 Implementing the Managed Resource

Before we implement our provider, we first need to implement the underlying managed resource. For a "real" provider this step is unecessary, since the resource already exists. We define the `Resource` class, which maintains a collection of memory-resident instances.

```
class Resource
{
public:
    Manager* manager;
    Array<Employee*> employees;
    Mutex mutex;

    Resource();
    ~Resource();
};
```

A `Resource` contains a single manager, an array of employees, and a mutex for synchronizing access to its instances. We declare a single global instance of this class as follows.

```
extern Resource resource;
```

All the providers presented below share this data structure. This is possible since all providers reside in this same library. The resource constructs when the library is loaded and destructs when it is unloaded. The constructor creates an instance of `Manager` and three instances of `Employee`.

```
Resource::Resource()
{
    Auto_Mutex am(mutex);

    Manager* m = Manager::create(true);
    m->Id.set(1001);
    m->First.set("Charles");
    m->Last.set("Burns");
    m->Gender.set(1);
    m->Active.set(true);
    m->NumEmployees.set(1037);
    m->Budget.set(10000000);
    manager = m;

    Employee* e;
    e = Employee::create(true);
    e->Id.set(4001);
    e->First.set("Homer");
    e->Last.set("Simpson");
    e->Gender.set(1);
    e->Active.set(true);
    employees.append(e);

    e = Employee::create(true);
    e->Id.set(4002);
    e->First.set("Carl");
    e->Last.set("Carlson");
    e->Gender.set(1);
    e->Active.set(true);
    employees.append(e);

    e = Employee::create(true);
    e->Id.set(4003);
    e->First.set("Lenny");
    e->Last.set("Leonard");
    e->Gender.set(1);
    e->Active.set(true);
    employees.append(e);
}
```

The destructor, destroys the memory-resident instances.

```
Resource::~Resource()
{
    Auto_Mutex am(mutex);

    Manager::destroy(manager);

    for (size_t i = 0; i < employees.size(); i++)
        Employee::destroy(employees[i]);
}
```

The provider implemented in this chapter is only concerned with the employee array.

6.2 Implementing the `load` Method

The `load` and `unload` methods are respectively called on provider load (start-up) and unload (shut-down). The load method contains any provider start-up tasks, such as:

- Initializing managed resources
- Opening files
- Creating threads
- Creating data structures

Our provider has nothing to do on load, so we leave the method empty as shown below.

```
Load_Status Employee_Provider::load()
{
    return LOAD_OK;
}
```

FNCT *before* `load` is called.

The CIM server can unload the provider at any time. Providers are unloaded under two conditions.

- When an arbitrary timeout expires.

- When the server shuts down.

The first condition is unavoidable but the second can be prevented by adding the following line to the `load` method.

```
cimom::allow_unload(false);
```

6.3 Implementing the `unload` Method

The `unload` method is called just before the provider is unloaded. This is where the provider performs shut-down tasks such as:

- Shutting down a managed resources
- Closing files
- Releasing threads
- Freeing data structures

Since our provider has nothing to do on unload, we leave this method empty as shown below.

```
Unload_Status Employee_Provider::unload()
{
    return UNLOAD_OK;
}
```

The resource instance is destructed *after* `unload` is called.

6.4 Implementing the `get_instance` Method

The `get_instance` method attempts to find an instance matching the `model` parameter, which specifies the keys as well as the required properties (signified by the non-null properties). Upon success, `instance` refers to the resulting instance. Our implementation searches the resource for a matching instance, as shown below.

```
Get_Instance_Status Employee_Provider::get_instance(
    const Employee* model,
    Employee*& instance)
{
    Auto_Mutex am(resource.mutex);

    for (size_t i = 0; i < resource.employees.size(); i++)
    {
        const Employee* e = resource.employees[i];

        if (key_eq(model, e))
        {
            instance = e->clone();
            return GET_INSTANCE_OK;
        }
    }

    return GET_INSTANCE_NOT_FOUND;
}
```

The `key_eq` function returns true if the two instances have identical keys. We use this function to check every employee instance for a match. If found, we set `instance` to the clone of the matching instance and return `GET_INSTANCE_OK`. Otherwise we return `GET_INSTANCE_NOT_FOUND`.

We mentioned above that the `model` parameter specifies the required properties. For example, the following snippet checks whether the `OutOfOffice` property is required.

```
if (!model->OutOfOffice.null)
{
    // Property is required.
}
```

Some providers use the property requirements to avoid unecesary property fetches. Our provider simply produces all properties, for simplicity.

If `get_instance` returns `GET_INSTANCE_UNSUPPORTED`, the adapter satisfies the request by calling `enum_instances` and searching for a matching instances. We recommend leaving `get_instance` unsupported when the total number of instances is small.

6.5 Implementing the `enum_instances` Method

The `enum_instances` method retrieves all instances of the given class. The `model` specifies the list of required properties (signified by the set of non-null properties). The `handler` is a callback object for delivering instances to the requestor. Our implementation delivers a clone of every employee in the resource, as shown below.

```
Enum_Instances_Status Employee_Provider::enum_instances(
    const Employee* model,
    Enum_Instances_Handler<Employee>* handler)
{
    Auto_Mutex am(resource.mutex);

    for (size_t i = 0; i < resource.employees.size(); i++)
    {
        Employee* e = resource.employees[i];
        handler->handle(e->clone());
    }

    return ENUM_INSTANCES_OK;
}
```

6.6 Implementing the `create_instance` Method

The `create_instance` method attempts to create a new instance. The `instance` parameter specifies zero or more property values of the new instance. You might expect the `instance` to specify values for all key properties, although this is not always so. Some providers assign keys values themselves. If so, the provider must update the keys of the `instance` parameter accordingly.

Our implementation first checks whether the instance already exists. If so it returns `CREATE_INSTANCE_DUPLICATE`. Otherwise it adds a clone of the instance to the employees array and returns `CREATE_INSTANCE_OK`.

```
Create_Instance_Status Employee_Provider::create_instance(
    Employee* instance)
{
    Auto_Mutex am(resource.mutex);

    for (size_t i = 0; i < resource.employees.size(); i++)
    {
        Employee* e = resource.employees[i];

        if (key_eq(instance, e))
            return CREATE_INSTANCE_DUPLICATE;
    }

    resource.employees.append(instance->clone());
    return CREATE_INSTANCE_OK;
}
```

6.7 Implementing the `delete_instance` Method

The `delete_instance` method attempts to delete the instance matching the `instance` parameter. Our implementation searches the resource for such an instance. If found, it removes and destroys it and returns `DELETE_INSTANCE_OK`. Otherwise it returns `DELETE_INSTANCE_NOT_FOUND`.

```
Delete_Instance_Status Employee_Provider::delete_instance(
    const Employee* instance)
{
    Auto_Mutex am(resource.mutex);

    for (size_t i = 0; i < resource.employees.size(); i++)
    {
        Employee* e = resource.employees[i];

        if (key_eq(instance, e))
        {
            resource.employees.remove(i);
            Employee::destroy(e);
            return DELETE_INSTANCE_OK;
        }
    }

    return DELETE_INSTANCE_NOT_FOUND;
}
```

6.8 Implementing the `modify_instance` Method

The `modify_instance` method attempts to modify an existing instance, which we call the *target*. The `model` parameter identifies the target instance and specifies which properties shall be modified. The `instance` parameter contains the new property values. For every non-null property of `model`, the corresponding property is copied from `instance` to the target instance. For example, the following fragment conditionally modifies the `Active` property.

```
if (!model->Active.null)
    target->Active = instance->Active;
```

This operation must be performed for each property. The `copy` function performs the above operation for every property as shown here:

```
copy(target, instance, model);
```

Our `modify_instance` implementation searches the array for a matching instance as shown below.

```
Modify_Instance_Status Employee_Provider::modify_instance(
    const Employee* model,
    const Employee* instance)
{
    Auto_Mutex am(resource.mutex);

    for (size_t i = 0; i < resource.employees.size(); i++)
    {
        Employee* e = resource.employees[i];

        if (key_eq(instance, e))
        {
            copy(e, instance, model);
            return MODIFY_INSTANCE_OK;
        }
    }

    return MODIFY_INSTANCE_NOT_FOUND;
}
```

If found, we modify it and return `MODIFY_INSTANCE_OK`. Otherwise we return `MODIFY_INSTANCE_NOT_FOU`

Chapter 7

Method Providers

This chapter adds two extrinsic methods to the instance provider developed in the previous chapter. Strictly speaking, there is no such thing as a "method provider" in CIMPLE. Formally, there are only three types of CIMPLE providers.

- definition - Implement the `get_instance` `enum_instances`, `create_instance`, `modify_instance`, and `delete_instance` methods
- definition - Implement the `references` `reference_names`, `association`, and `association_names` methods.
- definition - Provide for generation of indications and are enabled with the `enable_subscription` and `disable_association` methods

All three proovider typescan implement extrinsic methods. So when we informally refer to a *method provider*, we really mean one of these three types that happens to implement one or more extrinsic methods.

7.1 Extending the MOF Class

We begin by extending the MOF class definition introduced in chapter 6 by adding two extrinsic methods as shown below.

```
class Employee
{
    [Key] uint32 Id;
    string First;
    string Last;
    [Values{"Male", "Female"}, ValueMap{"1", "2"}]
    uint32 Gender;
    boolean Active = true;
    boolean OutOfOffice;

    uint32 SetOutOfOfficeState(
        [In]
        boolean OutOfOfficeState,
        [In(false), Out]
        boolean PreviousOutOfOfficeState);

    [Static] uint32 GetEmployeeCount();
};
```

The next section shows how to regenerate the source files to include these changes.

7.2 Regenerating the Sources

After changing the MOF class definition, we must:

1. Regenerate the class sources
2. Patch the provider sources
3. Regenerate the module source file

Again we use the `genproj` utility rather than running `genclass`, `genprov`, and `genmod` separately.

```
$ genproj Employee Employee
==== genclass:
Created Employee.h
Created Employee.cpp
created repository.h
Created repository.cpp
==== genprov:
Patched Employee_Provider.h
Patched Employee_Provider.cpp
==== genmod:
Created module.cpp
```

Since `Employee_Provider.h` and `Employee_Provider.cpp` already exist, `genprov` *patches* them. Patching updates intrinsic and extrinsic function signatures and inserts new extrinsic methods.

> **genprov and the end-marker**. If you generated your provider sources with a CIMPLE version prior to CIMPLE 1.0.0, then you must add an "end-marker" to the header file and source file where `genprov` will insert extrinsic methods. Do this by inserting the following line in both files..
>
> ```
> /*@END@*/
> ```
>
> You should also delete the `proc` function from the provider sources, since `genmod` now places it in `module.cpp`.

7.3 Implementing the `SetOutOfOfficeState` Method

The `SetOutOfOfficeState` implementation, shown below, first attempts to find an instance matching the `self` parameter (the instance on which the method is inovked). If found, it sets the `OutOfOffice` property, sets `PreviousOutOfOfficeState` to the previous value and returns 0. If not found, it returns 1 to signify and error.

```
Invoke_Method_Status Employee_Provider::SetOutOfOfficeState(
    const Employee* self,
    const Property<boolean>& OutOfOfficeState,
    Property<boolean>& PreviousOutOfOfficeState,
    Property<uint32>& return_value)
{
    Auto_Mutex am(resource.mutex);

    for (size_t i = 0; i < resource.employees.size(); i++)
    {
        Employee* e = resource.employees[i];

        if (key_eq(self, e))
        {
            PreviousOutOfOfficeState = e->OutOfOffice;
            e->OutOfOffice = OutOfOfficeState;
            return_value.set(0);
            return INVOKE_METHOD_OK;
        }
    }

    return_value.set(1);
    return INVOKE_METHOD_OK;
}
```

You might have noticed that this implementation has two kinds of return values.

- A *physical return value* – the return value of the C++ function, returned with the `return` statement. This must be one of the return values defined for CIMPLE Methods.
- A *logical return value* – the return value of the MOF method definition, returned in the method parameter parameter. This value must be consistent with the type of return value defined for the method in the mof.

The *physical return value* indicates whether the method is implemented or not (the unsupported method return is `INVOKE_METHOD_UNSUPPORTED`) . The complete set of C++ returns supported is in the following table:

Table 7.1: **CIMPLE Method Return Values**

C++ Return	**Description**
INVOKE_METHOD_OK	Method completes successfully
INVOKE_METHOD_FAILED	Method fails
INVOKE_METHOD_UNSUPPORTED	Method unsupported by the provider. This is the return defined when the provider template is created
INVOKE_METHOD_ACCESS_DENIED	Provider denies service for the request

7.4 Implementing the `GetEmployeeCount` Method

`GetEmployeeCount` is a static method. It is invoked on the class rather than on an instance of the class. Accordingly, there is no `self` member. The implemetation, shown below, simply returns the number of employees.

```
Invoke_Method_Status Employee_Provider::GetEmployeeCount(
    Property<uint32>& return_value)
{
    Auto_Mutex am(resource.mutex);

    return_value.set(resource.employees.size());
    return INVOKE_METHOD_OK;
}
```

7.5 Testing the Extrinsic Methods

The CIMPLE distribution provides an experimental tool called `ciminvoke`. This tool is a CIMPLE specific Pegasus client application used to invoke extrinsic methods. The following is an actual session used to test the two methods implemented in this chapter.

```
$ ciminvoke Employee.Id=4001 SetOutOfOfficeState OutOfOfficeState=true
return=0
PreviousOutOfOfficeState=false

$ ciminvoke Employee GetEmployeeCount
return=3
```

Chapter 8

Association Providers

This chapter shows how to develop an association provider. Association providers have the methods shown in the table below.

Table 8.1: **CIMPLE Association Provider Methods**

Association Provider Methods	**Required**
`load()`	no
`unload()`	no
`get_instance()`	no
`enum_instances()`	yes
`create_instances()`	no
`delete_instances()`	no
`modify_instances()`	no
`enum_associator_names()`	no
`enum_references()`	no

As indicated in column two, not all methods are required. Implementing just `enum_instances()` is sufficient for read-only association providers. When left unimplemented, the following methods are satisfied by calling `enum_instances()`.

```
    get_instance
    enum_associator_names
    enum_references
```

However, we recommend implementing these for large association sets in order to improve performance. But for smaller sets, they may be left unimplemented.

The provider presented in this chapter implements the `Link` association, which links a `Manager` to an `Employee`. The MOF definition is shown below.

```
[Association]
class Link
{
    [Key] Manager REF Mgr;
    [Key] Employee REF Emp;
};
```

We did not discuss the `Manager` instance provider but its source is included in the CIMPLE source distribution.

8.1 Implementing the `enum_instances` Method

The `Link` provider implements associations from a single manager (Charles Burns) to all instances in the resource. The `enum_instances()` implementation is shown below.

```
Enum_Instances_Status Link_Provider::enum_instances(
    const Link* model,
    Enum_Instances_Handler<Link>* handler)
{
    Auto_Mutex am(resource.mutex);

    for (size_t i = 0; i < resource.employees.size(); i++)
    {
        const Employee* e = resource.employees[i];

        Employee* emp = Employee::create(true);
        emp->Id = e->Id;

        Manager* mgr = Manager::create(true);
        mgr->Id.set(1001);

        Link* link = Link::create(true);
        link->Mgr = mgr;
        link->Emp = emp;

        handler->handle(link);
    }

    return ENUM_INSTANCES_OK;
}
```

By implementing this one method, we developed a complete read-only association provider. We now test it with the IndexOpenPegasus `cimcli` command.

```
$ cimcli an Manager.Id=1001
//redbird/root/cimv2:Employee.Id=4001
//redbird/root/cimv2:Employee.Id=4002
//redbird/root/cimv2:Employee.Id=4003
```

We do not show how to implement `create_instance()`, `delete_instance()`, and `modify_instance()` here, since these methods are covered in chapter 6 and their application to association providers is similar.

8.2 Implementing the `enum_associator_names` Method

This guide does not discuss the implementation of `enum_associator_names()`.

8.3 Implementing the `enum_references` Method

This guide does not discuss the implementation of the `enum_references()` method.

Chapter 9

Indication Providers

The functions to generate asychronous indications based on client subscriptions to define the indications to be delivered is a core capability of the CIM model.

Within the providers, there is a specific provider type that processes indications, the indication provider type. CIMPLE has implemented the Indication provider type and this chapter defines the rules for creating an indication provider with CIMPLE.

CIMPLE indication providers define the following methods.

Table 9.1: **CIMPLE Indication Provider Methods**

Indication Provider Methods
`load`
`unload`
`enable_indications`
`disable_indications`

A provider can generate indications either *passively* or *actively*.

- **Passive generation** is performed by an intrinsic or extrinsic provider method. In this case, the indication is generated in the thread used to call the method.
- **Active generation** is performed by a thread created by the indication provider.

9.1 Enabling and Disabling Indication Generation

In a CIM environment the generation of indications is controlled by the CIM Server through CIM Subscriptions that define the filters, indications, and destinations for indications. Providers are expected to generate indications only when the CIM Server has received the indication subscriptions and informed the proper providers that the subscriptions exist. Providers are notified that subscriptions with methods that enable or disble generation of particular indications.

CIMPLE provider indication generation is controlled by the following two method calls that are defined as part of the generation of each CIMPLE indication provider.

- `enable_Indications()` tells the provider to initiate the delivery of indications for the provider and provides information to the provider about the delivery of indications to the CIM Server
- **disable_Indications()** tells the provider to disable indication delivery.

Examples of the use of the `enable_indications` and `disable_indications` methods are shown in the sections below that define particular indication provider implementations.

9.1.1 Implementing the `enable_indications` Method

When the CIM Server determines that there are subscriptions for the indication provider, it calls the CIMPLE `enable_indications` method, whose prototype is defined below. This tells the provider that it may start generating indications. While there may be multiple subscriptions defined for a particular indication class and subscriptions may be created and destroyed within the CIM Server the indication provider is expected to remain active until the last subscription is destroyed in the CIM Server at which time, the CIM Server will send the `disable_indications` method.

The parameter provided with this mthod is `indication_handler` a pointer to the indication handler method defined for this Indication Class that is used to deliver indications from the provider to the CIM Server.

```
Enable_Indications_Status
<Classname>_Provider::enable_indications(
    Indication_Handler<Indication_Class>* indication_handler);
```

The provider MUST store the `indication_handler` and use it later to generate indications. The skeleton `enable_indications` method is generated by the CIMPLE utility `genprov` for any target class that is a has the Indication Qualifier and includes the code to save the `indication_handler` as shown in the examples in the following sections

The `enable_indications` method can be used by the provider writer to perform any other setup required to deliver indications. Note that in most servers, once the `enable_indications` method is called by the CIM Server, the provider is set to NOT unload until such time as the `disable_indications` method is received.

The `enable_indications` method has two possible returns:

- **ENABLE_INDICATIONS_OK** which is the normal response when the provider wants to accept the `enable_indications`.
- **ENABLE_INDICATIONS_FAILED** which tells the CIM Server that the provider does not accept the `enable_indications`. If the provider does not accept this method it should NOT try to deliver any indications AND should delete the handler.

9.1.2 Implementing the `disable_indications()` Method

The CIM server calls `disable_indication` when there are no longer any subscriptions to the indication defined by the indication provider.

This call tells the provider to stop generating indications. The provider MUST stop generating indications immediatly. It can use this method to stop any activity that is part of indication generation such as indication generation or polling threads, etc.

Also, The indication_handler that was provided with the `enable_indications`method MUST be deleted by the provider, normally in the `disable_indications` method.

The `Indication_Handler` is the only type of handler that the provider deletes.

The skeleton for this method is generated for the indication class when `genclass` is executed for the target class and the generated method includes the code to delete the `Indication_Handler` based on the location it was stored in the skeleton `enable_indications` method.

This method has two possible returns:

- **DISABLE_INDICATIONS_OK** which is the normal response when the provider wants to accept the `enable_indications`.
- **DISABLE_INDICATIONS_FAILED** which tells the CIM Server that the provider does not accept the `disable_indications`.

9.2 Delivering Indications to the CIM Server

The process of delivering indications to the CIM Server within a CIMPLE provider is a simple process of:

- Creating instances of the indication instance using the CIMPLE `create` method.
- Delivering the instances of the indication to the CIM Server using the `indication_handler` method provided by the `enable_indications` method.

The examples in the sections below demonstrate creation and delivery of indications.

9.3 `MyIndication` Indication Provider - Passive Generation

In this section we implement a simple indication provider that utilizes passive generation so that it generates indications when a CIM method defined by the provider is executed.

The classes for this example are as follows to define the indication class and a class to be embedded in the indication. The Indication class also includes a single method `DeliverIndications` that is called by a client to deliver an indication.

```
[Indication]
class MyIndication : CIM_Indication
{
    [EmbeddedObject]
    string object;

    [Static] uint32 DeliverIndications();
};

class MyEmbeddedClass
{
    string msg;
};
```

As explained in section 5.8, `genclass` generates the following C++ class for this mof.

```
class MyIndication : public Instance
{
public:
    // MyIndication features:
    Instance* MyEmbeddedClass;

    CIMPLE_CLASS(MyEmbeddedClass)
};
```

9.3.1 Implementing a passive `enable_indications` Method

When there are subscriptions for the `MyIndication` indication, the CIM server calls the `enable_indications` method, whose prototype is defined as follows.

```
Enable_Indications_Status enable_indications(
Indication_Handler<MyIndication>* indication_handler);
```

The provider should store the `indication_handler` to use later to deliver generated indications. The variable `_indication_handler` is defined in the adapter for this purpose.

The following is the operation method and is also exactly what is generated by the `genprov` utility.

```
Enable_Indications_Status MyIndication_Provider::enable_indications(
    Indication_Handler<MyIndication>* indication_handler)
{
    _indication_handler = indication_handler;

    return ENABLE_INDICATIONS_OK;
}
```

In this simple example, a method of the indication class is used to deliver indications. Each call to this method delivers a single indication using the `indication_handler` provided by the `enable_indications` method. Note that the inclusion of the `DeliverIndications` method implies that this provider is registered as both an indication and method provider.

```
Invoke_Method_Status MyIndication_Provider::DeliverIndications(
    Property<uint32>& return_value)
{
    return_value.value = 99;
    return_value.null = false;

    // The existence of _indication_handler is used to
    // determine  if indications are enabled.

    if (_indication_handler)
    {
        MyIndication* indic = MyIndication_create("777", "seven");
        _indication_handler->handle(indic);
    }

    return INVOKE_METHOD_OK;
}
```

Finally, we stop the provider from sending indications with the `disable_indications` method. In this provider, the only things that the `disable_indications` method must do are:

- Delete the `indication_handler` that was allocated as part of the `enable_indications` method.
- Flag to the indication generation mechanism that indication generation has been disabled. We do this by simply clearing the pointer to the `indication_handler`.

The complete code for this method is as follows. This is what is generated by the `genprov` utility as the skeleton for this method so that no code need be generated by the provider writer.

```
Disable_Indications_Status MyIndication_Provider::disable_indications()
{
    if (_indication_handler)
    {
        delete _indication_handler;
        _indication_handler = 0;
    }

    return DISABLE_INDICATIONS_OK;
}
```

This complete example is part of the CIMPLE working examples in the directory: /tt cimple/src/providers/MyIndications

9.4 `OutOfOfficeNotice` Indication Provider - Active Generation

In this section we consider how to implement an indication provider that utilizes active generation. An active provider creates a thread that publishes indications based on some criteria between the provider and its resources. In this example, we simply publish indications periodically while the indication provider is enabled.

Recall our discussion of embedded objects in section 5.8, where we first presented the following class.

```
[Indication]
class OutOfOfficeNotice
{
    [EmbeddedObject]
    string employee;
};
```

As explained in section 5.8, `genclass` generates the following C++ class.

```
class OutOfOfficeNotice : public Instance
{
public:
    // OutOfOfficeNotice features:
    Instance* employee;

    CIMPLE_CLASS(OutOfOfficeNotice)
};
```

The class generator converts the `employee` string property to an `Instance` pointer. Otherwise, the provider would have to encode the employee as a string (either in XML or MOF).

9.4.1 Implementing the active `enable_indications` Method

As soon as there are subscriptions for the `OutOfOfficeNotice` indication, the CIM server calls the `enable_indications` method, whose prototype is defined as follows.

```
Enable_Indications_Status
OutOfOfficeNotice_Provider::enable_indications(
    Indication_Handler<OutOfOfficeNotice>* indication_handler);
```

The provider should store the `indication_handler` and use it later to generate indications. The handler should be deleted by the `disable_indications` method (the `Indication_Handler` is the only type of handler that the provider should delete). The following snippet generates an indication using the handler.

```
OutOfOfficeNotice* notice;
.
.
.
indication_handler->handle(notice);
```

Our `enable_indications` implementation, shown below, saves the indication handler and creates a thread that periodically generates indications.

```
Enable_Indications_Status
OutOfOfficeNotice_Provider::enable_indications(
    Indication_Handler<OutOfOfficeNotice>* indication_handler)
{
    // Save indication handler.
    _indication_handler = indication_handler;

    // Create indication thread. It is created as a joinable thread
    // so that it can be joined when the disable_indications method
    // is called as a positive indication that the thread stopped.
    // The this parameter passes the provider pointer to the thread.
    _continue.inc();
    Thread::create_joinable(
        _thread, (Thread_Proc)_indication_thread, this);

    return ENABLE_INDICATIONS_OK;
}
```

The `OutOfOfficeNotice_Provider::continue` member, defined below, is an *atomic counter* (see 11.5 for more information on the atomic counter functions) used to signal the thread to exit.

```
Atomic_Counter _continue;
```

We increment it to 1 before creating the thread. The thread exits when this counter becomes zero. The `Thread::create_joinable` function creates a joinable thread that runs `_indication_thread`, defined below.

The thread function shown below generates and delivers an OutOfOfficeNotice indication for each employee instance that has OutOfOffice status and then sleeps for a second. each indication contains the embedded Employee instance This continues while the provider is enabled as defined by the continue variable.

```
void* OutOfOfficeNotice_Provider::_indication_thread(void* arg)
{

    // recover the provider pointer from input arg

    OutOfOfficeNotice_Provider* provider =
        (OutOfOfficeNotice_Provider*)arg;

    // loop to create indications while continue nonzero
    while (provider->_continue.get())
    {
        // lock the resource to prevent changes during the
        // search and generation of indications
        resource.mutex.lock();

        for (size_t i = 0; i < resource.employees.size(); i++)
        {
            const Employee* e = resource.employees[i];

            if (e->OutOfOffice.value)
            {
                OutOfOfficeNotice* notice =
                    OutOfOfficeNotice::create(true);
                notice->employee = clone(e);
                provider->_indication_handler->handle(notice);
            }
        }

        resource.mutex.unlock();

        // Sleep 1 second between after generating indications
        Time::sleep(1 * Time::SEC);
    }

    return 0;
}
```

This method scans the resource every second and generates indications for employees that are out of office. The thread loops as long as `_continue` is non-zero. When it becomes zero, the thread function exits.

9.4.2 Implementing the Active disable_indications Method

The CIM server calls `disable_indication` when there are no longer any subscriptions to the `OutOfOfficeNotice` indication.

This method performs the following steps.

- Signals the indication thread to exit.
- Joins with the indication thread.
- Deletes the indication handler.

Our implementation is shown below.

```
Disable_Indications_Status
OutOfOfficeNotice_Provider::disable_indications()
{
    // Destroy indication thread.
    _continue.dec();

    //join the indication thread as an indication that
    // it is finished
    void* value_ptr;
    Thread::join(_thread, value_ptr);

    // Delete indication handler.
    delete _indication_handler;
    _indication_handler = 0;

    return DISABLE_INDICATIONS_OK;
}
```

The implementation of this provider is available as one of the CIMPLE examples in the CIMPLE source directory:

`cimple/src/providers/Employee`.

Note that there are other functions within CIMPLE that might be used to manage the indication thread such as the scheduler described in 11.7.

Chapter 10

Logging And Tracing

Starting with version 1.2.0 CIMPLE provides functions to standardize generation of log entries independent of any particular provider interface so that log entries can be defined once when the code is created. The log functions send logs to a log file. A resource file can be defined for the user to control output of log information.

The logging facility consists of:

- a log API to create log entries
- 5 levels of log severity that can be selectively output
- a set of macros to simplify adding log entries to providers
- a resource file that defines what is to be logged

The logging facility is defined in more detail in the following subsections.

10.1 Overview

Because there is no standard logging facility interface defined for providers CIMPLE provides a mechanism so that log entries can be defined in the code and output in a manner independent of any particular platform. Today this mechanism uses a single log file for all cimple providers.

If logging is enabled (cimple configured with the `--enable\_debug option`) the log file is opened when the first provider starts and log calls are selectively output to the log file depending on the definition of log severity defined in a cimple resource file. To enable logging in the generated providers define :

TBD: THIS IS NOT CORRECT.

```
$ ./configure --enable-debug ...
```

All entries are output with:

- Information about location in the source,
- time,
- provider identification,
- additional information provided with the log call

10.2 The log API and Macros

Providers use the logging by calling the `log` function defined in the `cimple/log.h` header file. For example:

```
log(LL_DBG, __FILE__, __LINE__, "my name is %s; my age is %d", "John", 12);
```

There are five log levels:

- LL_FATAL
- LL_ERR
- LL_WARN
- LL_INFO
- LL_DBG

Shortcut macros are provided for each log level to simplify the invocation of `log`. For example:

```
    CIMPLE_DBG(("my name is %s; my age is %d", "John", 12));

    CIMPLE_DBG(("modify_instance key = %u", instance->Key.value));
```

The macros all have the same form:

```
CIMPLE_XXX((<format string>, <parameter list>));
   where: <format string> is a standard printf C++ definition of
              the ouput format
          <parameter list> is a list of variables that corresponds to the
              items in the format string
          XXX is last part of the the log level name as defined below.
```

The defined macro names are:

- CIMPLE_FATAL
- CIMPLE_ERR
- CIMPLE_WARN
- CIMPLE_INFO
- CIMPLE_DBG

Note that the double parenthensis are REQUIRED because of the variable number of arguments in the macro.

10.3 The CIMPLE Resource file and logging

A CIMPLE resource file controls what is written to log files. This is a single file in the users home directory with the name `.cimplerc`

This file must contain a line that sets the logging level. For exmaple:

```
LOG_LEVEL=DBG
```

/noindent This defines the minimum level of log that is output. This level and all higher levels are output. So if `WARN` is set, `WARN`, `ERR`, and `FATAL` log entires are output.

`LOG_LEVEL` must be set to one of the following.

- FATAL
- ERR
- WARN
- INFO
- DBG

The following is an example of this file:

```
: cat ~/.cimplerc
LOG_LEVEL=DBG
```

Note that today the log-level is the only attribute in this resource file.

10.4 The log file

CIMPLE logs are written to a single log file located in a special directory named .cimple under the users home directory. New messages are appended to a file named `messages`.

CIMPLE itself has no provisions to clean or delete this file.

The log entries have formats similar to the following where the datetime, provider file, and line number are standard and the last part is defined as part of the log entry:

```
2009/02/13 09:56:06 DBG: All\_Class_Provider.cpp(31): get\_instance key = 9999
2009/02/13 09:56:06 DBG: All\_Class_Provider.cpp(140): delete\_instance key = 9999
```

10.5 Logging Information from CIMPLE Adapters

In addition to provider generated log entries, the CIMPLE adapters themselves make log entries for warnings and errors during the normal operation of CIMPLE. We recommend configuring with debug and watching for log messages during provider development.

NOTE: Currently the CMPI provider logs information that is controlled through the same resource file.

```
$ ./configure --enable-debug ...
```

The following would be some examples of logs from the CMPI Adapter:

```
2009/02/13 09:56:07 DBG: CMPI\_Adapter.cpp(2067): enter: cleanup()
2009/02/13 09:56:07 DBG: CMPI\_Adapter.cpp(2078): return: cleanup(): CMPI\_RC\_OK
2009/02/13 09:56:07 DBG: CMPI\_Adapter.cpp(240): enter: ~CMPI-_Adapter()
```

Chapter 11

Multi-Thread Programming

CIMPLE providers inherently allow multi-threading. The calls to providers from most CIM Servers are multi-thread where each new operation call occurs on a different thread and there may be multiple calls to a provider outstanding simultaneously.

Also, the characteristics of many provider operations such as indication generation or resource management may require the existence of multiple threads in the provider to accomplish their functions.

There are a number of concepts required to provide effective multi-thread programming including thread creation and termination, thread synchronization (joins and blocking), scheduling, atomic and condition variables.

CIMPLE includes mechanisms for:

- Creating and managing Threads. Section 11.1
- Thread Specific Data. Section 11.2
- Mutexes and AutoMutexes. Section 11.3
- Atomic Counters(Atomic Integers). Section 11.5
- Condition Variables. Section 11.4
- Condition Queues. Section. Section 11.6
- Scheduling thread execution. Section 11.7

To support multi-threading in a portable manner independent of the characteristics of the particular CIM Server and OS, CIMPLE provides a set of threading APIs and classes that can be used with all CIMPLE adapters and OS's to manage threads.

The characteristics of these multi-threading functions are based on POSIX pthread equivalent mechanisms. These CIMPLE classes, in fact, normally wrap corresponding pthread functions on platforms where pthreads is available. However, this set of classes brings a single portable threading API set common to all CIMPLE supported platforms whether they support pthreads or not. This means that the CIMPLE developed provider even with multithreading functions uses the same source code when moved from platform to platform and from adapter to adapter.

The thread APIs are defined in the following header files in the `src/cimple` directory:

Table 11.1: **Multi-threading Header Files**

Header File	Functions
`Thread.h`	Define the Thread Management functions. Section 11.1
`Mutex.h`	Define Mutexes to provide exclusive locking of threads. Section 11.3
`TSD.h`	Utilize thread-specific data. Section 11.2
`AutoMutex.h`	Define AutoMutexes based on Mutexes that automatically destruct when they go out of scope. Section 11.3
`Cond.h`	Define Condition Variables. Section 11.4
`Atomic_Counter.h`	Define Atomic Integers that can be safely and atomically manipulated by multiple threads. Section 11.5
`Cond_Queue.h`	Define Condition Queues that allow synchronizing work between queues. Section 11.6
`Scheduler.h`	Define Scheduler to execute functions at specific times or intervals. Section 11.7

The following sections describe the threading APIs available with CIMPLE. There are examples of the common usage of these functions integrated into the CIMPLE tests in the directory `src/cimple/tests`.

11.1 The Thread Class

CIMPLE includes a Thread class defined in the header `Thread.h` that defines a common mechanism for implementing multi-thread programming in the provider.

The CIMPLE Thread class defines a set of methods parallel to the POSIX pthreads environment to allow creation and management of threads and are common across all of the CIMPLE supported environments. The functions are defined as follows:

Table 11.2: **Thread Class Methods**

Method	Description
Thread Constructor	Create a new thread object. Ex. `Thread _thread;`
create_joinable	Static function to create a new joinable thread.
join	Static function to wait for the defined thread to terminate and return the threads return value.
create_detached	Static method to create a detached thread.
exit	Static method to terminate the calling thread and define a value to be returned to joining thread.
self	Static method to get the calling threads identity.
equal	Static method to compare two thread Ids for equality

The following simple example creates a number of threads and then waits for them to join the original thread. Each thread executes a function that waits one second and then exits the threaded function.

First we define the function to be executed in a thread. To keep the implementation simple we have provided for a single `void*` argument for the thread function in which you can place what you want and extract it in the thread. The following example simple sleeps one second and returns the input argument.

```
// Thread process function that sleeps for one second and
// then exits the thread.
static void* _proc(void* arg)
{
    char* str = (char*)arg;
    Time::sleep(Time::SEC);
    Thread::exit(arg);
    return arg;
}
```

Next we create the function to create and start the thread (in this case a joinable thread and once the thread is started to wait for the thread to complete an join the calling thread. The other code in this example is simply to define an arg parameter and test that it was returned with the join.

```
// Create a set of threads and wait for them to join this thread
// before terminating.
int main(int argc, char** argv)
{
    Thread my_thread;

    // create and start a joinable thread to call _proc
    int r1 = Thread::create_joinable(my_thread, _proc, (void *)"abc");

    // test that thread was created
    assert(r1 == 0);

    void* value = 0;
    int r2 = Thread::join(my_thread, value);

    // test that join worked
    assert(r2 == 0);

    // test return value
    assert(strcmp((char*)value,"abc") == 0);

    return 0;
}
```

An example of programming threads in a CIMPLE provider to create indications is shown in section 9.1.1 (Implementing the enable_Indications Method).

11.2 Thread Specific Data (TSD)

This section not complete.

11.3 Mutex and AutoMutex Classes

CIMPLE provides an implementation of mutexes `Mutex` and automutexes `AutoMutex` as classes in the header files `Mutex.h` and `Auto_Mutex.h`. This implementation provides portability of these functions throughout all of the OSs that CIMPLE supports. It implements selected functionality of the POSIX mutexs wrapped in the CIMPLE Mutex class.

11.3.1 Mutexes

The CIMPLE Mutex implementation allows the creation of both recursive and non-recursive mutexes. Mutex is a single class with the following methods:

Table 11.3: **Mutex Methods**

Method	**Description**
Constructor	Create a Mutex. Ex. `Mutex _mutex;`
lock	Lock the mutex. If the mutex is recursive the `lock()` may be applied repeatedly and the mutex does not unlock until an unlock() call has been executed for each lock() call. Ex. `_mutex.lock();`
try_lock	Lock the mutexMutex!try_lock if unlocked otherwise return error code (new CIMPLE 2.0.12). If the mutex is recursive the `trylock()` may be applied repeatedly and the mutex does not unlock until an unlock() call has been executed for each lock() call. Ex. `_mutex.try_lock();`
unlock	`Unlock()` the mutex. If the mutex is recursive, it does not unlock until the number of `unlock()` calls matches the number of `lock()` calls. Ex. `_mutex.unlock();`

```
int main(int argc, char** argv)
{
    Mutex m;
    m.lock();
    m.unlock();
    return 0;
}
```

11.3.2 AutoMutexes

The `Auto_Mutex` class defined in the header file `Auto_Mutex.h` is used to automatically lock an existing mutex upon construction of the `Auto_Mutex` and unlock it upon destruction (when the `Auto_Mutex` goes out of scope). The `Auto_Mutex` is particularly useful where there are many places a mutux should be unlocked and especially with exceptions.For example:

```
    static Mutex _mutex;    .
    .
    .
    void foo()
    {
        // Lock mutex here!
        Auto_Mutex auto_mutex(_mutex);
        . . .
        . . .
        //mutex unlocks here as Auto_Mutex goes out of scope!
    }
```

11.4 Condition Variables Class (Cond)

The CIMPLE class `Cond` implements a conditional variable, similar to the POSIX-threads conditional variable. A conditional variable is used with appropriate functions for waiting and thread continuation. It allows a thread to suspend execution and relinquish the processor until a defined condition is true. The CIMPLE condition varaible is defined in the file `Cond.h`.

While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.

A condition variable is a synchronization object that allows a single thread or multiple threads to block their execution until a resource enters a selected state. Condition variables are usually associated with a predicate (boolean) which indicates the state. A thread which waits on a condition variable should test the predicate within the exclusion zone provided by the associated mutex to determine the resource state.

A condition variable must always be associated with a mutex to avoid a race condition created by one thread preparing to wait and another thread which may signal the condition before the first thread actually waits on it resulting in a deadlock. The thread will be perpetually waiting for a signal that is never sent. Any mutex can be used, there is no explicit link between the mutex and the condition variable except the `wait()` method.

NOTE: Proper locking and unlocking of the associated mutex variable is essential when using condition variables. For example:

- Failing to lock the mutex before calling `wait()` may cause it NOT to block.
- Failing to unlock the mutex after calling `signal()` may not allow a matching `wait()` routine to complete (it will remain blocked).

Condition variables may be signaled or broadcast. A signaled condition variable unblocks a single waiting thread while a broadcast condition variable unblocks all waiting threads.

Spurious wakeups may occur when waiting on a condition variable. A spurious wakeup occurs when a thread returns from a condition wait when it should really continue waiting. A normal signal being delivered to a thread may cause a spurious wakeup during a condition wait. Since the return values from `wait()` do not imply anything about the value of the predicate, the predicate should be re-evaluated.

Signals are not remembered, which means that threads must already be waiting for a signal to receive it. The signal/broadcast will be lost if there are no waiting threads.

The heart of condition varaible behavior is the `wait()` function. When executed this function first unlocks the defined mutex and then waits on a signal or broadcast from another thread. The thread is sleeping. When a signal is received the waiting thread will initially relock the defined mutex. When the waiting thread has acquired the lock it will return from the `wait()` call allowing the thread to continue execution.

Generally the pattern for condition variable usage is as follows:

```
Cond cond_variable;
Mutex condition_mutex;
bool condition;

waiter_thread()
{
    condition_mutex.lock();
    while(!condition)
        condition_mutex.wait(condition_mutex);
    do_something_within_the_lock();
    condition_mutex.unlock();
    ... continue;
}

signaler_thread()
{
    ...
    condition_mutex.lock();
    condition = true;
    cond_variable.signal();
    condition_mutex.unlock();
    ...
}
```

The CIMPLE conditional variable class is defined in `Cond.h` and includes the following methods
The following code is a simple example of a condition variable operating in conjunction with a Mutex so that a `take_action()` function is executed only when a count decrements to zero. It defines an action thread that waits on the condition and a counter function called from some other thread that decrements a counter

Table 11.4: **Conditional Variable Methods**

Method	Description
Constructor	Create a Condition variable. Ex. `Cond cond1;`
signal	Wake up one of the threads that is blocking on the wait() method. Ex. `cond1.signal();`
broadcast	Wake up all of the threads that are blocking on the wait() method(new CIMPLE 2.0.12). Ex. `cond1.signal();`
wait	Block until a thread calls signal(). The mutex provided as the input parameter is unlocked while the thread is waiting and relocked upon wakeup. Ex. `cond1.wait(mutex1);`

```
// The condition variable
Cond cond_var;
Mutex condition_mutex;
int count = Some Initial value;

// action waits on condition variable.
// take_action() is executed when counter() is called enough times
// so that count decrements to zero.
action_thread()
{
    . . .
    // Lock the condition mutex to avoid race conditions
    condition_mutex.lock();
    // Give up control until condition variable == 0
    while (count <> 0)
        condition_mutex.wait(condition_mutex);
    condition_mutex.unlock();
    take_action();
    . . .
}

// counter_function decrements the counter on each call.
counter_function()
{
    ...
   // lock the mutex, then decrement and test counter
   // the lock is used to avoid race conditions
    condition_mutex.lock();
    count--;
    // When count goes to zero, signal the condition variable
    if (count == 0)
```

11.5 Atomic Counter Class

CIMPLE includes the implementation of an Atomic Counter function that is defined in the header `Atomic_Counter.h`. The Atomic Counter represents an int and provides thread-safe atomic operations on the integer.

The `Atomic_Counter` includes the following methods.

Table 11.5: **Atomic Counter Methods**

Method	**Description**
default Constructor	Create an Atomic Counter variable
Initializing Constructor	Create an Atomic Counter and sets the value
get	get the value of the Atomic Counter
inc	Increment the Atomic Counter
dec	Decrement the value of the Atomic Counter
dec_and_test	Decrement the value of the Atomic Counter and returns true if it is set to zero after the decrement

An example of the use of the Atomic Counter to control a thread is shown in section 9.1.1 (Implementing the enable_Indications Method).

11.6 Condition Queues Class

The CIMPLE condition queue mechanism is defined by a class in the file `Cond_Queue.h`

This class provides a thread-safe queue implementation so that threads may exchange data. The `dequeue()` function blocks until an entry is available. The `enqueue()` method blocks until there are less than `max_size elements`.

The class includes the following methods:

Any information can be enqueued through the single void* argument that is passed through the `enqueue()` and available to the `dequeue()` methods.

The following is a simple example of the use of a condition queue to pass information between a reader and writer. This and other examples can be seen in the test programs in `src/cimple/tests`.

Table 11.6: **Condition Queue Methods**

Method	Description
`constructor()`	Creates a condition queue
`destructor()`	Destroys a condition queue
`enqueue()`	Add an entry to a condition queue
`dequeue()`	Removes an entry from a condition queue

```
// writer writes NUM_ENTRIES entries to the queue

static void* _writer(void* arg)
{
    Cond_Queue* queue = (Cond_Queue*)arg;
    for (size_t i = 0; i < NUM_ENTRIES; i++)
        queue->enqueue((void*)i);
    return 0;
}

// Reader dequeues entries up to NUM_WRITES
static void* _reader(void* arg)
{
    // get the condition queue from the arg parameter.
    Cond_Queue* queue = (Cond_Queue*)arg;

    for (size_t i = 0; i < NUM_WRITES; i++)
    {
        void* entry = queue->dequeue();
        assert(size_t(entry) == i);
        printf("reader: %zu\n", (size_t)entry);
    }
    return 0;
}
int main
{
    // code to execute the writer and reader functions

    // create the condition queue, max_size 1
    Cond_Queue queue(1);

    // Create reader thread with condition queue as the arg parameter
    Thread thread;
    Thread::create_joinable(thread, _reader, &queue);

    // call the writer to enqueue entries.
    _writer(&queue);
```

11.7 The Scheduler Class

See the header file `Scheduler.h` for more information on the scheduler.

CIMPLE includes a scheduler that allows the user to schedule functions with a defined time delay. The scheduler class allows the user to setup work to be executed in the future, either once or repetitively and on a defined schedule.

The scheduler can handle multiple items in its queue and initiates them as they are scheduled. The timer for the scheduler operates at the microsecond level although the actual resolution of the timers is dependent on the operating system.

The scheduler is defined in a single class, Scheduler and includes the following methods:

Table 11.7: **Scheduler Class Methods**

Method	**Description**
`constructor`	Create a scheduler object
`add_timer`	Add an entry to a scheduler to be executed at a defined time in the future
`remove_timer`	Remove an item from the scheduler queue
`dispatch`	Process the scheduler queue. Can be used in place of the `start_thread` and `stop_thread` or the automatic thread control defined below to process schedule items on the current thread.
`start_dispatcher`	Start an independent thread that controls scheduling by calling the dispatcher. This function is used only if the automatic dispatcher option is not used.
`stop_dispatcher`	Stop the scheduler dispatch thread if it has been started.
`clean`	Clean any uncompleted scheduled work out of a stopped scheduler.

If the user wishes to make an item repetitive(schedule the function at regular intervals) this can be controlled by the return from the scheduled function. If the function defined by `add_item(..)` returns zero, this is considered one-shot. The item is discarded after the schedule is executed. If the function returns an integer not equal to zero, this is considered the new schedule time and the item is restarted with this time.

The schedule has 3 modes of operation.

- **Manual** The user calls the `dispatch()` directly in a loop to process a Schedule queue. In this case the Scheduler operates in the current thread.
- **Manual Separate Thread** The user executes the `start_dispatcher()` function to start a new thread in which the scheduler process schedule items until the `stop_dispatcher()` function is called. This new thread continues to operate whether there are scheduled items in the scheduler queue or not.
- **automatic Dispatcher thread** The user simply adds schedule items to the scheuduler queue. When there are items in the queue, the scheduler dispatches them on a thread controlled by the dispatcher. When there are no items on the scheduler queue, the scheduler thread is terminated. The user can stop the scheduler with `stop_dispatcher()`

The Scheduler executes schedule items on a single thread whichever mode of operation is used. When a scheduled item expires, it is executed and the `dispatch()` function waits for it to complete before initiating the next item.

NOTE: If you schedule tasks that will take extensive time, this will disrupt the orderly dispatching of scheduled items.

The following is a example of simple scheduling using the automatic scheduler option:

First we create a function to be scheduled. This is simply a function with a single argument that will be called from the scheduler dispatch function. Since all such functions are scheduled in sequence, this function should not lock up the thread upon which it is called.

The single argument allows you to define anything you want to pass to this function. The following example simply gets the time from the argument and uses it to return.

The return from a scheduled function is an unsigned integer that defines for the scheduler whether to create a new item in the scheduler or not. If the return is zero, no new item is created. If the return is non-zero a new item is created with the delay defined by the return value.

```
// function to be scheduled
static uint64 _scheduled_function(void* arg)
{
    // get the next schedule time from arg
    uint64 time = (uint64)atoi(arg);

    ... // do secheduled function

    // tell dispatc()h to schedule next event in time usec.
    return (uint64)time;
}
```

The following code is also a scheduled function. This one stops the scheduler when it is executed.

```
// When executed, this function stops the scheduler

static uint64 _stopScheduler(void* arg)
{
    sched.stop_thread();
    return 0;
}
```

Finally we can create create some scheduled events for the scheduler with calls to `add_timer()`. Each timer defines the following 3 parameters:

- The interval in milliseconds until this item expires and the scheduled function is called.
- The function to be called.
- The single argument as a `void*` that will be passed to the scheduled function.

The following code defines the scheduler itself (in this case a scheduler that automatically controls the thread for its dispaching) and then schedules 2 items to be called in 3 seconds and 9 seconds respectively. Note that the `_scheduled_function(...)` defined above, restarts these scheduled items when it executes to that they are repetitive, executing once each 3 and 9 seconds respectively.

A third scheduled item, calls the function `_stopScheduler()` function after 30 seconds to stop the schedule dispatcher and shut down the scheduler.

Finally, since the schedule dispatcher is executed as a joinable thread, we can determine when the scheduler is finally shut down by executing a join of the scheduler thread. Once this join is complete, the scheduler thread has stopped.

Finally we clean out any remaining items in the scheduler queue although the destructor would also do this.

```
// Create a scheduler that automatically manages the
// dispatcher thread.
Schedule sched(true);
int main(int argc, char** argv)
{
    // add two scheduled functions,, one in 3 sec and one
    // in 9 sec. These will call _scheduled_function()

    sched.add_timer(3 * SEC, _scheduled_function, (void*)3);
    sched.add_timer(9 * SEC, _scheduled_function, (void*)9);

    // call the stop thread function after 30 seconds.
    sched.add_timer(30 * SECOND, _stopScheduler, (void*)"Nothing");

    // wait for the dispatcher thread to rejoin when the stopthread
    // function runs.
    void * value;
    Thread::join(sched.thread_id(), value);

    // clean the scheduler queue.
    sched.clean();
    return 0;
}
```

Chapter 12

CIMServer Upcalls

The CIM Server is normally capable of providing:

- Accessing instances or methods of other classes in the server or other providers.
- Selected information about the operation itself
- Selected controls over the provider itself.

However, each CIMServer and provider interface has a unique interface for these upcalls. Thus, for example, the Pegasus C++ call is through a set of operations similar to the Pegasus client operations (getInstance, etc.) that operate againse a cimomHandle supplied when the provider is initialized.

To create a standard interface that can be used independent of any particular CIM Server or provider interface CIMPLE has implemented mechanisms to access this information from within the CIMPLE provider.

All of these interfaces are defined in a single header file `cimple/cimom.h`.

12.1 Accessing Instances of Other Classes in the CIM Server

Many providers need information from other providers or may need to execute methods on these providers. While most CIM Servers allow upcalls in one form or another, the mechanism is different for each provider interface and, using the CIM Server services would not return CIMPLE instances but instances in the native data format of the server.

Therefore CIMPLE provides an upcall mechanism that executes upcalls to the server for instances in a standard portable manner and uses CIMPLE definitions for the classes and instances directly.

This mechanism supports the following upcalls:

- enum_instances
- get_instance
- create_instance
- delete_instance
- modify_instance
- invoke_method

The upcall mechanism does not provide upcalls for references and associations for several reasons:

- To date the functionality has not been required. The primary reason for the upcalls has been to a) get information to create association instances (typically enumerateInstances or enumerateInstanceNames or b) get properties from instances of other classes to insert into instances of the target class for this provider.
- Many of the CIM Servers today have issues with executing reference and associator upcalls from providers in general because of the multiplicity of provider calls they generate.

Note that a provider may acquire the same information provided by an association or reference upcall normally by making an enumerateInstances upcall on the association class.

The upcall mechanisms are defined in detail in subsequent sections.

12.1.1 Defining the Class for an upcall

The class to be used for an upcall is defined in a manner similar to the target class for the provider. You must make the Class mof available to repository.mof at the time of the genclass so that that metadata for the class is generated.

However, no provider is created for this class so that the upcall class is NOT included in the getprov execution. A simple example is the Upcall provider in the test providers directory. It is based on a single Class (Upcall) which expects to enumerate instances of the CIM_ComputerSystem class.

The mof might look like the following in repository.mof:

```
#pragma include ("Upcall.mof")
```

The upcall class is not included in repository.mof because it is available from the cim model itself which is avaible to genclass through environment variable definitions.

The provider generation looks like:

```
> genclass Upcall CIM_ComputerSystem
> genprov Upcall
> genmod Upcall Upcall
```

Or when built into a Makefile, the target definitions might look like:

```
regmod:
$(BINDIR)/regmod -c $(TARGET)

genclass:
$(BINDIR)/genclass -r Upcall CIM\_ComputerSystem

genprov:
$(BINDIR)/genprov Upcall

genmod:
$(BINDIR)/genmod Upcall Upcall
```

To use the upcall class within the target provider you must only declare the class with a call like the following example:

```
    // Create the CIMPLE model for CIM_ComputerSystem

    CIM_ComputerSystem* model = CIM_ComputerSystem::create();
```

this `model` variable is not available to be used with any of the upcall methods defined below.

Note that all upcall methods are part of the cimom C++ class.

12.1.2 Upcall Common Characteristics

All of the upcall operations have the following common characteristics:

All upcall operations include a return with:

- 0 = success
- 1 = failure

All of the upcall operations include the same first parameter, the namespace upon which the operation is to be executed.

In all of the calls, the second parameter defines the instance or object path that is the target of the request. This is defined through a CIMPLE instance representing the class or path of the target.

The remaining parameters are dependent on the call as defined in the following sections.

12.1.3 Enumerate Instance Upcalls

The Enumerate instance enumerates instances of the defined class and its subclasses.

Enumerate instances is unique in that it generates multiple instances in its response.To easily accomodate the multiple object response and also to allow for the future of the cimom where response may not be monolithic (all objects requested in a single call, this operation includes the definition nof an iterator to extract the responses.

The call is shown in the following example:

```
CIM_ComputerSystem* model = CIM_ComputerSystem::create();
// Define the instance enumerator
Instance_Enumerator ie;

// Call the enumerate function for CIM_ComputerSystem
if (cimom::enum_instances("root/cimv2", model, ie) != 0)
    return -1;
else
{
    // loop to iterate all response CIMPLE instances
    for (; ie; ie++)
    {
        // get the next iteration form the iterator
        Ref<Instance> inst = ie();

        // cast to an instance of CIM_ComputerSystem
        CIM_ComputerSystem* ccs =
             cast<CIM_ComputerSystem*>(inst.ptr());

        print(ccs);
    }
 }
```

The main characteristics of the iterator are:

- The enumeration constructor
- Inclusion of the constructor in the enum_instances call
- Definition of the iteration loop. Defined with the variable name from the constructor which returns a boolean. This is conceptually equivalent to the iterator method more().
- Pulling the next CIMPLE instance from the iterator. This is conceptually equivalent to the iterator method next() and returns a REF of the next instance.
- Casting the instance to the target class. This is required to make the meta-model accesible for the returned instance.

All instances are returned as Ref¡Instance¿ so that their scope is the current block.

The prototype for the enumeration operaton is:

```
static int enum_instances(
    const char* name_space,
    const Instance* model,
    Instance_Enumerator& enumerator);
```

12.2 Get Instance Upcalls

The prototype for the enumeration operaton is:

```
static Ref<Instance> get_instance(
    const char* name_space,
    const Instance* model)
```

12.2.1 Create Instance Upcalls

Not documented in this version of the document.

12.2.2 Delete Instance Upcalls

Not documented in this version of the document.

12.2.3 Modify Instance Upcalls

Not documented in this version of the document.

12.2.4 Invoke Method Upcalls

Not documented in this version of the document.

12.3 Accessing information on the current Operation

12.4 T

here may be several pieces of information about the current operation that the provider needs such as user name. The existence of this information and the form of the request is varys widely by provider interface. Therefore CIMPLE has implemented a standard mechansim to request this information.

- Providing other information on the operation such as user name

NOTE: Today the user name is the only information available through this interface.

While an operation is current, information on the current user may be requested with an upcall as follows:

```
String userName;

if(cimom::get_user_name(userName))
{
    ... code to use the name. Note the name itself may be empty.
}
else
{
    ... No user name was available from this server
}
```

12.5 Controlling the Provider

Some servers attempt to minimize memory usage by forcing providers out of memory if they are idle for extended periods of time. For example, typically Pegasus forces providers out of memory if they are not used for several minutes and are not active indication providers. At least on some servers this behavior can be modified by informaing the server that the provider does not want to be unloaded.

There is an upcall in `cimom.h` that allows the provider to request the server to disable unloading the provider.

This is demonstrated as follows:

```
// disable unload for this provider
cimom::allow_unload(false);

.....

// allow provider unload for this provider
cimom::allow_unload(true);
```

Chapter 13

Instance Map Class (memory cache)

The Instance Map class provides a memory based container that allows a number of general and `CIMPLE` CIM operation specific operations functions to be performed against a set of instances in the container.

NOTE: This class is not optimized for large numbers of instances and uses a linear search to find instances in the array.

This class is defined in the CIMPLE header `Instance_Map.h`. .

This is a template based class so that defining a class with `genclass` provides the basis for the general and specific Instance Map functions for that class.

The general functions manipulate the instances in the container. They allow insertion, extraction, and removal of individual instances in the container and cleaning of the complete container. The CIM operation specific operations allow the provider writer to directly substitute the use of the instance map container into the definiton of individual CIMPLE CIM operations in place of manually building and managing the instances as part of the operation.

The local functions include:

- `insert` - Insert an instance into a map if it does not already exist.
- `size` - Return the number of instances in the map.
- `find` - Find an instance in a map and return an index to its location
- `lookup` - Return an instance from a map if the instance exists in the map
- `remove` - Remove an instance from a map.
- `clear` - Clear the entire map
- `[ ... ]` - return the instance at the defined index.

Instance maps are defined in terms of the CIM Object path for the instances so that the searches associated with find(), lookup(), etc. are all executed based on the key properties.

To provide support for the `CIMPLE` instance operations, additional functions are defined so that the user can define access to the map for create, modify, delete, get and enumerate instances with a single line of code using instances in a map. The class includes the C++ Class templates required to directly implement the functions of:

- `get_instance` - get the defined instance from the map and return it to the CIMServer
- `enum_instances` - get the instances from the map and return to them the CIMServer
- `create_instance` - Create an instance and insert it into the map
- `delete_instance` - Delete an instance from the map
- `modify_instance` - Modify an instance in the map

/noindent With a single line of code in the corresponding `CIMPLE` CIM Operation.

The instance map operations return exactly the same return status codes that `CIMPLE`return status codes defines for the corresponding `CIMPLE`CIM Operations CIM Operations. Therefore, these functions can be used directly within the CIM Operations. Thus, for example, the map `get_instance` operation returns `GET_INSTANCE_NOT_FOUND` if the requested instance is not in the map so that this status can be passed directly to the CIM Server.

As an example, the following code would completely implement the `enum_instances` `CIMPLE` CIM operation using instances maintained in an instance map.

```
Enum_Instances_Status CMPL_Base_Provider::enum_instances(
    const CMPL_Base* model,
    Enum_Instances_Handler<CMPL_Base>* handler)
{
    // User code to get instances from a predefined map
    return _map.enum_instances(model, handler);
    return ENUM_INSTANCES_OK;
}
```

Note that the only line of user code is the map statement.

13.1 Defining and building a CIMPLE Instance Map

A CIMPLE instance map is a memory based cache for a single class. It is defined through the map constructor for the class. The Instance Map definition is an extension of Array. In the following example, the instance map is defined in the CMPL_Base_Provider.h module by adding the Instance_Map constructor for the CMPL_Base class.

```
// Generated by genprov 2.0.8
#ifndef _CMPL_Base_Provider_h
#define _CMPL_Base_Provider_h

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

CIMPLE_NAMESPACE_BEGIN

class CMPL_Base_Provider
{
public:
. . .
private:
    // User added constructor for Instance Map
    Instance_Map<CMPL_Base> _map;
. . .
};
```

Instances installed in the map with the insert function as shown in the following example and the normal instance create functions of CIMPLE. In this case, because this is an artifical example, the instances are created as part of the load functions. It would be expected that in a working provider the management of instances in the map would be part of the management of the resource (i.e. create, delete, modify of instances in the map as the provider manages the real resource).

```
Load_Status CMPL_Base_Provider::load()
{
    CMPL_Base* instance;

    // create and insert an instance
    instance = CMPL_Base::create();
    instance->Key.value = 1;
    instance->info.value = "First Instance";
    _map.insert(instance);

    instance = CMPL_Base::create();
    instance->Key.value = 2;
    instance->info.value = "Second Instance";
    _map.insert(instance);
    return LOAD_OK;
}
```

This example has created two instances and places them in the map with the insert function.

13.2 Instance Map Manipulation Functions

The `Instance_Map` provides the methods defined in table 13.1 to manipulate instances in an instance map based on the existence of a `CIMPLE` Instance Map for the defined class within the provider.
We have already show above how an instance can be inserted into the map. In addition, the following example shows testing for the existence of an instance, getting an instance, removing an instance, and clearing the map.

```
    CMPL_Base* instance;

    // create an instance key
    model = CMPL_Base::create();
    model->Key.value = 1;

    size_t index;
    // get the number of instances in the map
    size_t size = _map.size();

    // get the instance defined by model if it exists
    CMPL_Base* rtn;
    if((rtn = _map.lookup(model)) != 0);
       ... use the instance returned in rtn

    // or
    if ((index = _map.find(instance)) != (size_t)-1)
        CMPL_Base* rtn_instance = _map[index];

    // remove the instance from the map
    if ((index = _map.find(instance)) != (size_t)-1)
    _map.remove(index);

    // Clear the map and destroy any instances in the map
    _map.clear();
```

13.3 Implementing Instance Map based CIMPLE CIM Operations

This class represents the implementation of an for CIMPLE. Through this class a provider can be written where all manipulation of the instances for the class is managed asynchronously to the CIM Operations.

Since the map functions directly parallel the corresponding CIM Operations, the complete set of CIM Operations can be directly implemented so that CIM create, get, enumerate, modify, and delete operations are directed to the instance map for instances.

The instances are maintained in the Instance map and the instance map class provides both operations to managed these instances and operations so that the CIM Operations on these instances directly access the instances in the map. Thus, the CIM Operations for create, modify, get, enumerate, and delete directly interface with instances in the instance map and asynchronously the provider can create, modify, delete instances in the instance map.

The following example of `CIMPLE` CIM operations implements all of the operations for the `CMPL_Base` class in terms of instances in the instance map for this class. This is implemented in the sample provider in the `src/providers/DerivedAssoc` provider in the `CIMPLE`source tree source tree.

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

Enum_Instances_Status CMPL_Base_Provider::enum_instances(
    const CMPL_Base* model,
    Enum_Instances_Handler<CMPL_Base>* handler)
{
    // User added call to map to get instances
    return _map.enum_instances(model, handler);
    return ENUM_INSTANCES_OK;
}

Create_Instance_Status CMPL_Base_Provider::create_instance(
    CMPL_Base* instance)
{
    // map directly handles creation of instances
    return _map.create_instance(instance);
}

Delete_Instance_Status CMPL_Base_Provider::delete_instance(
    const CMPL_Base* instance)
{
    map function directly deletes instances
    return _map.delete_instance(instance);
}

Modify_Instance_Status CMPL_Base_Provider::modify_instance(
    const CMPL_Base* model,
    const CMPL_Base* instance)
{
    // map function directly modifies instances
    return _map.modify_instance(model, instance);
}
```

This represents a complete implementation of the `CMPL_Base` class where the instances are completely managed aschronously to the CIM Operations and cached in memory.

Table 13.1: **Instance Map Manipulation Methods**

Function Name	**Description**
`constructor`	Insert an instance into a map if it does not already exist. Return position inserted or -1 if the instance already exists. Pt. `Instance_Map<CLASS> <map_name>;` Ex. `Instance_Map<CMPL_Base> _map;`
`size`	Return the number of instances in the map. Pt. `size_t <map_name>.size();` Ex. `size_t pos = _map.size();`
`find`	Find an instance in a map with equal keys and return the index to its location. Return -1 if the instance is not found. Pt. `size_t <map_name>.find(Instance_model*);` Ex. `size_t pos = _map.find(_model);`
`lookup`	Return the pointer to an instance from a map if the instance exists in the map. Return value 0 for the instance pointer if the instance cannot be found Pt. `<ClassName>  <map_name>.lookup(Instance_model*);` Ex. `size_t pos = _map.lookup(_model);`
`remove`	Remove an instance from a map. If the index is out of range, return with no change made.l Pt. `void <map_name>.remove(Instance_model*);` Ex. `_map.remove(_model);`
`clear`	Clear the entire map and `destroy()` the instances. Pt. `void <map_name>.clear();` Ex. `_map.clear();`
`[ index ]`	Return the instance at the defined index. If the index is out of range, the returned instance pointer is set to 0. Pt. `<ClassName>*  <map_name>[const size_t index];` Ex. `CMPL_Base* = _map[index];`

Chapter 14

Other APIs in the CIMPLE environment

14.1 Stacks Class (Experimental)

There is a `Stack` class defined in `src/cimple/stack.h` but today it should be considered experimental.

14.2 Lists Class (Experimental)

There is a simple `list` class defined in the header `src/cimple/List.h` that can be used as the basis for creating doubly linked lists as an extension to the defined class.

See the header file for more information.

This class should be considered experimental

Chapter 15

Registering and Installing Providers

Provider registration and installation today is specific to each supported platform. In this section we detail the registration mechanisms for each of the supported platforms.

The process of registration and installation involves the following steps

- Moving the provider library to the location required by the CIM Server.
- Installing the CIM class or classes for which the provider is defined. Note that this may involve installing superclasses for the defined classes or updating an earlier version of the target class(es).
- Registration of the provider which means telling the CIM Server about the provider including, library, Classes/functions handled, possible security, etc.

Of course this may also involve security and file permissions issues depending on the installation.

15.1 Registering/Installing Providers for Pegasus

15.1.1 Automatic Provider Registration for Pegasus

Pegasus registers providers by installing instances of 3 Pegasus specific classes into the Pegasus repository(PG_ProviderModule, PF_Provider, PG_ProviderCapabilities). Within the native Pegasus environment, provider registration is done by manually creating the instances of these three classes and installing the instances with either /verb—cimmof— or /verb—cimmofl—. It is significantly safer and easier if this process can be automated.

Appendix D contains the MOF file you would have to write to register our `President` provider.

CIMPLE has automated the registration process for the Pegasus CIMServer with a utility `regmod`.

CIMPLE `regmod` automates provider registration for Pegasus including

- Building the instances of the three classes required to register a provider. `regmod` uses information from the provider shared library as parameters for this registration
- Installing the classes used by the provider and any required superclasses into the running server
- Copying the provider library to the location defined by the server

For example, the following command registers and installs all providers contained in `libPresident.so`.

```
$ regmod libPresident.so
Using CMPI provider interface
Registering President_Provider (class President)
```

`regmod` performs all of the required steps including:

- Creating the registration instances using information from the shared library.
- Installing the provider classes and the instances of the registration instances into the running server. Note that regmod REQUIRES that the server be running because this insures that the validation facilities of the server are used to properly validate the classes and instances.
- Copying the provider shared library to the server location for provider libraries.

The `-c` option creates any classes the provider module uses that are not already in the Pegasus repository. For our provider, it creates the `President` class the first time it runs.

Sometimes you may need the `regmod -d` option that dumps the MOF registration instances required to register the provider, without actually registering anything or modifying the Pegasus repository. For more on the `regmod` tool type:

```
regmod -h
```

15.2 Registering Providers for SFCB

The registration process for SFCB involves placing the information into a special directory known to an SFCB utility and then using this utility to compile any classes and instances.

The actual registration of providers is based on an ASCII file defined for SFCB that provides registration information in a name/value format. While it requires much of the same information as Pegasus registration the form of the file is completely different.

Today providers for SFCB must be registered manually (CIMPLE `regmod` does not provide support for SFCB registration. The process is typically as follows:

- Copy the Provider shared library to the location required by SFCB. Typically this is `/usr/lib` or `/usr/lib64`
- Copy the provider MOF to the SFCB staging directory
- Create a provider registration file for SFCB. We must manually create this file. This file defines the provider name, location, type, and namespace for SFCB
- Copy the provider registration file to the SFCB staging directory
- execute the SFCB script `sfcbrepos` to install the provider and classes

The following is an example of SFCB registration from the widget provider supplied with the CIMPLE distribution

```
## These locations may change per users needs
SFCB_MOFSDIR=/usr/local/var/lib/sfcb/stage/mofs/root/cimv2
SFCB_REGSDIR=/usr/local/var/lib/sfcb/stage/regs
SFCB_NAMESPACE=root/cimv2

## Note that we are forcing copy of target to lib64 for now.
## Currently there is an issue with use of other directories for sfcb
## at least in some Linux platforms.
register-sfcb:
cp $(TARGET) /usr/lib64/
cp CIMPLE_Widget.mof $(SFCB_MOFSDIR)/CIMPLE_Widget.mof
cp CIMPLE_Widget.reg $(SFCB_REGSDIR)/CIMPLE_Widget.reg
sfcbrepos
```

The SFCB registration file for Widget is as follows:

```
[CIMPLE_Widget]
  provider: CIMPLE_Widget_Provider
  location: cimplewidget
  type: instance method
  namespace: root/cimv2
```

15.3 Registering and Installing Providers for WMI

WMI is different that the previous registration mechanisms in that it uses special qualifiers to define much of the registration information.

Providers for WMI must be registered manually (not directly supported by `regmod`). The process is typically as follows:

First we must copy the provider DLL to the WMI providers directory, usually located here:

```
C:\windows\system32\wbem\
```

Second we use the WMI MOF compiler `mofcomp` to add our classes to the CIM repository as shown below.

```
mofcomp repository.mof
```

Third we register our provider as follows with mofcomp.

```
mofcomp register.mof
```

Finally, we register our WMI provider as a COM server:

```
regsvr32 /s C:\windows\system32\wbem\Person.dll
```

In the examples provided with CIMPLE, these operations are integrated into the `Makefile` provided as the target register.

The following is an example of SFCB registration from the CIMPLE test providers for the Person provider (`wmi/person/Makefile`):

```
register:
SOURCEPATH=$(subst /,\,$(TARGET))
DESTPATH=c:\WINDOWS\system32\wbem\$(LIBRARY).dll

register: install
mofcomp -N:root/cimv2 providerclasses.mof
mofcomp -N:root/cimv2 register.mof
regsvr32 /s $(DESTPATH)

install:
copy $(SOURCEPATH) c:\WINDOWS\system32\wbem

## Removes the service registration
## Does not remove the repository or registration
## mof.
unregwmi:
regsvr32 /u /s $(DESTPATH)
```

Chapter 16

CIMPLE WMI Providers

CIMPLE supports use of CIMPLE providers acting as WMI providers in a Microsoft Windows environment. The CIMPLE provider code and APIs used for other CIM Servers should function correctly in the Windows environment as a WMI provider. An adapter is provided to adapt CIMPLE objects to WMI objects and interface between CIMPLE and WMI operations.

CIMPLE provides support for WMI instance operations, WMI indication generation, and WMI extrinsic methods.

16.1 Microsoft Operating Systems and Compilers

CIMPLE has been tested with a number of Micrososft OSs and Microsoft Visual Studio environments to insure that it is portable across these environments. CIMPLE WMI provider should compile and run under at least the following Microsoft environments:

Operating Systems:

- Windows XP
- Windows Server 2003
- Windows Vista

Microsoft Visual Studio versions:

- Visual Studio .NET 2003(known also as Visual C++ 7.1)
- Visual Studio 2005(known also as Visual C++ 8.0)
- Visual Studio 2008(known also as Visual C++ 9.0)

In all cases our testing is done with the latest service packs for each OS and Visual Studio Release.

16.2 Special Characteristics of WMI Providers

There are a number of special characteristics involved in running providers under WMI including:

- Extra Libraries for the provider link and a link definition file.
- Special qualifiers for the provider class definition to support registration of the provider.
- Special classes required by Microsoft as superclasses for indications in place of the DMTF Indication class.
- WMI does not support the association operations so that the references and association functions are not used.

16.2.1 Extra Qualifiers

WMI uses special qualifiers to register providers `dynamic` and `implemented` as part of the class definition process.

They are not part of the currently defined qualifier set in the DMTF specifications. Therefore, you must add them to the qualifier definitions in qualifiers.mof in the CIM mof schema to get the class definition to compile with `genclass`. They have been inserted in the schemas provided. The definitions that must be inserted are:

```
qualifiers.mof:Qualifier Dynamic : boolean = false,
qualifiers.mof:Qualifier Implemented : boolean = false,
```

These qualifiers must be added to the provider classes as follows:

- `dynamic` - Class level provider that defines this class as a class for which the instances are provided dynamically. The Dynamic qualifier must be specified on all classes that contain data and for which instances are created dynamically. The Provider qualifier is typically also specified to identify the provider responsible for supplying the data. Classes that contain only methods that need implementation do not require the Dynamic qualifier. Only the Provider qualifier is required to specify the name of the provider to supply the implementation.All classes derived from a dynamic class must be dynamic. You cannot derive a static class from a dynamic class
- **implemented** - Indicates that a method has an implementation supplied by a provider

- **provider** - Defines the provider associated with a dynamic class

The **provider** qualifier should be part of the existing DMTF qualifier definitions either in `qualifiers.mof` or `optionalqualifiers.mof`. The value for this qualifier MUST match the name of the provider DLL (without the extension).

The easiest way to do this and still keep the definitions as general as possible is to a) create a special mof file that contains these qualifiers and include this file in repository.mof so that these qualifiers are available for provider class compilation.

repository.mof

```
// Includes both the WMI qualifiers and the
// classes required for this provider
#pragma include ("wmiqualifiers.mof")
#pragma include ("ProviderClasses.mof")
```

wimiqualifiers.mof

```
// The following qualifiers are required for WMI providers
//
// Added for Windows. Required to register providers
Qualifier Dynamic : boolean = false,

    Scope(class, association, indication);

// Added for Windows.
Qualifier PropertyContext : string = Null,

    Scope(property);

// Added for Windows.
Qualifier Implemented : boolean = false,

    Scope(method);

[indication]
class __ExtrinsicEvent
{
};
```

person.mof

```
// Person class defined compatible with creation and registration
// of WMI provider. Uses the dynamic, provider, and implemented
// qualifiers.

[dynamic, provider("Person")]
class Person
{
    [Key] string SSN;
    string FirstName;
    string LastName;

    [implemented, static]
    uint32 foo([in] string arg);
};
```

Typically other CIM Servers should not object to these new qualifiers inclusion in the class definitions for the provider. However, since the provider qualifier may be used by other CIM Servers (except Pegasus) this qualifier should be treated as WMI only in the provider class definitions.

16.2.2 WMI Provider Linking

Linking requires extra windows libraries `ole32.lib` and `oleaut32.lib`. These must be include in the link definition.

Linking requires an extra file, the `link.def` file that is defined to the linker through the `/def:` option (ex. `/def:link.def`.

This file has the form:

```
LIBRARY "Person.dll"
EXPORTS
    DllMain PRIVATE
    DllCanUnloadNow PRIVATE
    DllGetClassObject PRIVATE
    DllRegisterServer PRIVATE
    DllUnregisterServer PRIVATE
```

The example below shows a `Makefile` for the Person provider:

```
TOP=../../..
include $(TOP)/mak/config.mak
PROVIDER_MODULE = Person
Classes = Person
LIBRARY = Person
SOURCES = \
    Person.cpp \
    Person_Provider.cpp \
    module.cpp \
    repository.cpp
LIBRARIES = cimplewmiadap cimple

## Extra link flag required by wmi
EXTRA_LINK_FLAGS = /def:link.def
EXTRA_SYS_LIBS = ole32.lib oleaut32.lib

## set to use wmi adpater
DEFINES += -DCIMPLE_WMI_MODULE

include $(TOP)/mak/rules.mak
```

16.2.3 WMI Provider Registration and Installation

This section shows how to register a WMI provider using the Microsoft tools.

WMI provider registration and installation includes the following steps:

- Copying the provider shared library to the location required by Microsoft, typically `c:`
  `WINDOWS`
  `system32`
  `wbem`.
- Compiling the provider classes with the Microsoft MOF compiler (`mofcomp`.
- Compiling the registration MOF `register.mof` which was created by `genclass` with `mofcomp`.
- Registering the shared library with
  tt regsvr32.

First we must copy the provider DLL to the WMI providers directory, Usually located here:

```
C:\windows\system32\wbem\
```

Second we use the WMI MOF compiler to add our classes to the CIM repository as shown below.

```
mofcomp repository.mof
```

Third we register our provider as follows.

```
mofcomp register.mof
```

Finally, we register our WMI provider as a COM server:

```
regsvr32 /s C:\windows\system32\wbem\Person.dll
```

In the examples provided with cimple, these operations are integrated into the Makefile provided as the target reg.
The following example is the component of a `Makefile` for registration and installation of a provider into the WMI environment.

```
## provider registration, installation and removal targets
## normally user should only need to call register
#
register:
SOURCEPATH=$(subst /,\,$(TARGET))
DESTPATH=c:\WINDOWS\system32\wbem\$(LIBRARY).dll

register: install
mofcomp -N:root/cimv2 providerclasses.mof
mofcomp -N:root/cimv2 register.mof
regsvr32 /s $(DESTPATH)

## Remove the service registration
## Does not remove the repository or registration
## mof.
unregwmi:
regsvr32 /u /s $(DESTPATH)

install:
copy $(SOURCEPATH) c:\WINDOWS\system32\wbem

restartwmi: stop start

# stop WMI server
stop:
net stop winmgmt

# start WMI server
start:
net start winmgmt
```

16.2.4 Special Classes

One significant difference with Windows is the superclass for indications.

While the superclass for all indications in DMTF is Indications, the superclass for indications in WMI is __**ExtrinsicEvent**. Thus, a new indication class would be defined as subclasses from the __**ExtrinsicEvent** class

```
[dynamic, indication, provider("GadgetProvider")]
class Buzzer : __ExtrinsicEvent
{
    [Key] string key;
    string message;

    [implemented, static]
    uint32 trigger();
};
```

16.3 Example of a WMI Provider

This example explains how to build a trivial provider for WMI. For the most part, it is like building a CIMPLE provider for other servers, but there are a few minor differences in the definition of the mof, linking and registration.

NOTE: This provider is included in the distribution as an example.

16.3.1 Creating the Person WMI Provider

We start with the following MOF definition (which we place in `providerclasses.mof`). This example is part of the cimple distribution in in the directory `cimple/src/wmi/person`.

```
[dynamic, provider("Person")]
class Person
{
    [Key] string SSN;
    [Key] string FirstName;
    [Key] string LastName;

    [implemented]
    uint32 foo([in] string arg);
};
```

Then we add the definitions for the `dynamic` and `implemented` qualifiers to a file `wmiqualifiers.mof`.

The definitions that must be inserted are:

```
qualifiers.mof:Qualifier Dynamic : boolean = false,
qualifiers.mof:Qualifier Implemented : boolean = false,
```

We must include this file in `repository.mof` as follows:

```
// Includes both the wmi qualifiers and the
// classes required for this provider
#pragma include ("wmiqualifiers.mof")
#pragma include ("ProviderClasses.mof")
```

Next we use the `genproj` command to generate the classes, provider, and module. The individual utilities (`genclass`, `genprov`, and `genmod` can also be used.

```
C:\> genproj Person Person
Created Person.h
Created Person.cpp
created repository.h
Created repository.cpp
Created Person_Provider.h
Created Person_Provider.cpp
Created module.cpp
Created guid.h
Created register.mof
```

The generated files are as follows:

- **Person.h** - the Person class declaration
- **Person.cpp** - the Person class definition
- **repository.h** - the class repository declarations
- **repository.cpp** - the class repository definitions
- **Person_Provider.h** - the Person provider declaration
- **Person_Provider.cpp** - the Person provider methods
- **module.cpp** - the WMI entry points
- **guid.h** - the GUID that uniquely identifies the provider COM server
- **register.mof** - the WMI registration instances

Note that when compiling for WMI, the CIMPLE utilities generate two extra files `guid.h` and `register.mof`. Whereas in the general case, registration information is generated by `regmod` since some of this information is required for the build and link process for WMI providers, it is generated by `genproj`.

16.3.2 Compiling and Linking the Person Provider

Next we must compile and link the provider. A link definition file `link.def` must be created as shown below.

```
LIBRARY "Person.dll"

EXPORTS
        DllMain PRIVATE
        DllCanUnloadNow PRIVATE
        DllGetClassObject PRIVATE
        DllRegisterServer PRIVATE
        DllUnregisterServer PRIVATE
```

Then we create the following `Makefile`.

```
## TOP defines location of the cimple mak directory
TOP=../../..
include $(TOP)/mak/config.mak

LIBRARY = Person
SOURCES = Person.cpp Person_Provider.cpp module.cpp repository.cpp
LIBRARIES = cimplewmiadap cimple
EXTRA_LINK_FLAGS = /def:link.def
EXTRA_SYS_LIBS = ole32.lib oleaut32.lib
DEFINES += -DCIMPLE_WMI_MODULE

include $(TOP)/mak/rules.mak

## provider registration, installation and removal targets
register:
SOURCEPATH=$(subst /,\,$(TARGET))
DESTPATH=c:\WINDOWS\system32\wbem\$(LIBRARY).dll

register: install
mofcomp -N:root/cimv2 providerclasses.mof
mofcomp -N:root/cimv2 register.mof
regsvr32 /s $(DESTPATH)

## Remove the service registration
## Does not remove the repository or registration
## mof.
unregwmi:
regsvr32 /u /s $(DESTPATH)

install:
copy $(SOURCEPATH) c:\WINDOWS\system32\wbem

restartwmi: stop start

# stop WMI server
stop:
net stop winmgmt

# start WMI server
start:
net start winmgmt
```

Finally, we build the provider as shown below.

```
C:\> make
```

This creates a DLL called **Person.dll**.

16.3.3 Registering the Person WMI provider

The following Makefile segment defines registration for our Person Provider.

```
register: install
mofcomp -N:root/cimv2 providerclasses.mof
mofcomp -N:root/cimv2 register.mof
regsvr32 /s $(DESTPATH)

## Remove the service registration
## Does not remove the repository or registration
## mof.
unregwmi:
regsvr32 /u /s $(DESTPATH)

install:
copy $(SOURCEPATH) c:\WINDOWS\system32\wbem

restartwmi: stop start

# stop WMI server
stop:
net stop winmgmt

# start WMI server
start:
net start winmgmt
```

We register the provider and install it with a single make operation

```
make register
```

In the examples provided with cimple, these operations are integrated into the Makefile provided as the target reg.

16.3.4 Verifying the Person WMI provider

There several tools available help verify the WMI provider once it is installed including:

- **cimbrowser.exe** - Part of a wmi toolset available from Micrososft under the name CIMTest. This is a complete graphic WMI CIM browser.
- **wbemtest.exe** - Client program that executes wmi CIM operations from a set of check boxes.

Either of these tools is helpful to verify the providers you write with CIMPLE. You may also use many other WMI client tools or build command line test tools with one of the Microsoft scripting mechanisms.
In any case, to verify this first provider you should confirm that the Person class was installed in the property namespace (normally `Root/cimv2`) and that the provider returns two instances of the Person class.
You can validate the response from the defined method as follows:

Appendix A

Code Complexity Comparisons

This appendix compares the complexity of various source code implementations done with these three provider interfaces: CIMPLE, Pegasus, CMPI.

A.1 Creating an Instance

The following subsections show how create and instance of the **President** class using the following provider interfaces: CIMPLE, Pegasus, and CMPI.

A.1.1 With CIMPLE

```
President* inst = President::create(true);
inst->Number.set(1);
inst->First.set("George");
inst->Last.set("Washington");
```

A.1.2 With Pegasus

```
try
{
    Array<CIMKeyBinding> bindings;
    bindings.append(CIMKeyBinding("Number", "1", CIMTYPE_UINT32));
    CIMObjectPath path("President");
    path.setKeyBindings(bindings);
```

```
        CIMInstance inst("President");
        inst.setPath(bindings);
        inst.addProperty(CIMProperty("Number", Uint32(1)));
        inst.addProperty(CIMProperty("First", String("George")));
        inst.addProperty(CIMProperty("Last", String("Washington")));
    }
    catch (Exception& exception)
    {
        // Handle exception.
    }
```

A.1.3 With CMPI

```
CMPIStatus status;
CMPIValue value;
CMPIObjectPath* path;
CMPIInstance* inst;

path = CMNewObjectPath(broker, NULL, "President", &status);

if (status.rc != CMPI_RC_OK)
{
    /* Handle error */
}

value.uint32 = 1;
CMAddKey(path, "Number", &value, CMPI_uint32);

inst = CMNewInstance(broker, path, &status);

if (status.rc != CMPI_RC_OK)
{
    /* Handle error */
}

value.uint32 = 1;
status = CMSetProperty(inst, "Number", &value, CMPI_uint32);
```

```
if (status.rc != CMPI_RC_OK)
{
    /* Handle error */
}

value.string = CMNewString(broker, "George", &status);

if (status.rc != CMPI_RC_OK)
{
    /* Handle error */
}

status = CMSetProperty(inst, "First", &value, CMPI_string);

if (status.rc != CMPI_RC_OK)
{
    /* Handle error */
}

value.string = CMNewString(broker, "Washington", &status);

if (status.rc != CMPI_RC_OK)
{
    /* Handle error */
}

status = CMSetProperty(inst, "Second", &value, CMPI_string);

if (status.rc != CMPI_RC_OK)
{
    /* Handle error */
}
```

A.2 Implementing a Simple Extrinsic Method

A.2.1 With CIMPLE

```
Invoke_Method_Status Adder_Provider::add(
    const Adder* self,
    const Property<real64>& x,
    const Property<real64>& y,
    Property<real64>& return_value)
{
    return_value.value.set(x.value + y.value);
    return INVOKE_METHOD_OK;
}
```

A.2.2 With CMPI

```
CMPIStatus TestCMPIMethodProviderInvokeMethod(
    CMPIMethodMI* mi,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* ref,
    char* methodName,
    const CMPIArgs* in,
    CMPIArgs* out)
{
    CMPIStatus status = { CMPI_RC_OK, NULL };

    /* Handle add() method. */

    if (strcasecmp(methodName, "add") == 0)
    {
        unsigned int n;
        unsigned int i;
        CMPIData data;
        CMPIString* name;
        CMPIReal64 x = 0.0;
        int foundX = 0;
        CMPIReal64 y = 0.0;
```

```
int foundY = 0;
CMPIReal64 z;
CMPIValue sum;

/* Check number of arguments. */

n = CMGetArgCount(in, &status);

if (status.rc != CMPI_RC_OK)
    return status;

if (n != 2)
{
    status.rc = CMPI_RC_ERR_FAILED;
    return status;
}

/* Get x and y parameters. */

for (i = 0; i < n; i++)
{
    data = CMGetArgAt(in, i, &name, &status);

    if (status.rc != CMPI_RC_OK)
        return status;

    if (strcasecmp(CMGetCharPtr(name), "x") == 0)
    {
        if (data.type != CMPI_real64)
        {
            status.rc = CMPI_RC_ERR_TYPE_MISMATCH;
            return status;
        }

        x = data.value.real64;
        foundX = 1;
        continue;
    }
```

```
            if (strcasecmp(CMGetCharPtr(name), "y") == 0)
            {
                if (data.type != CMPI_real64)
                {
                    status.rc = CMPI_RC_ERR_TYPE_MISMATCH;
                    return status;
                }

                y = data.value.real64;
                foundY = 1;
                continue;
            }

            status.rc = CMPI_RC_ERR_INVALID_PARAMETER;
            return status;
        }

        /* Be sure we got both x and y. */

        if (!foundX || !foundY)
        {
            status.rc = CMPI_RC_ERR_FAILED;
            return status;
        }

        /* Add */

        sum.real64 = x + y;

        /* Add output parameter. */

        CMReturnData (rslt, (CMPIValue*)&sum, CMPI_real64);
        CMReturnDone (rslt);
        return status;
    }

    /* Method not found. */
```

```
    status.rc = CMPI_RC_ERR_METHOD_NOT_FOUND;
    return status;
}
```

Appendix B

The President Provider Skeleton

This appendix includes the source code generated by the following command.

```
$ genprov President
Created President_Provider.h
Created President_Provider.cpp
```

B.1 President_Provider.h

```
#ifndef _President_Provider_h
#define _President_Provider_h

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

CIMPLE_NAMESPACE_BEGIN

class President_Provider
{
public:

    typedef President Class;

    President_Provider();
```

```
    ~President_Provider();

    Load_Status load();

    Unload_Status unload();

    Get_Instance_Status get_instance(
        const President* model,
        President*& instance);

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

    Create_Instance_Status create_instance(
        President* instance);

    Delete_Instance_Status delete_instance(
        const President* instance);

    Modify_Instance_Status modify_instance(
        const President* model,
        const President* instance);
};

CIMPLE_NAMESPACE_END

#endif /* _President_Provider_h */
```

B.2 President_Provider.cpp

```
#include "President_Provider.h"

CIMPLE_NAMESPACE_BEGIN

President_Provider::President_Provider()
{
}
```

```
President_Provider::~President_Provider()
{
}

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

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

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

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

Create_Instance_Status President_Provider::create_instance(
    President* instance)
{
    return CREATE_INSTANCE_UNSUPPORTED;
}

Delete_Instance_Status President_Provider::delete_instance(
    const President* instance)
{
```

```
    return DELETE_INSTANCE_UNSUPPORTED;
}

Modify_Instance_Status President_Provider::modify_instance(
    const President* model,
    const President* instance)
{
    return MODIFY_INSTANCE_UNSUPPORTED;
}

CIMPLE_NAMESPACE_END
```

Appendix C

The President Provider Implementation

This appendix includes the source listing for the President provider described in chapter 3.

C.1 President_Provider.h

```
#ifndef _President_Provider_h
#define _President_Provider_h

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

CIMPLE_NAMESPACE_BEGIN

class President_Provider
{
public:

    typedef President Class;

    President_Provider();

    ~President_Provider();
```

```
    Load_Status load();

    Unload_Status unload();

    Get_Instance_Status get_instance(
        const President* model,
        President*& instance);

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

    Create_Instance_Status create_instance(
        President* instance);

    Delete_Instance_Status delete_instance(
        const President* instance);

    Modify_Instance_Status modify_instance(
        const President* model,
        const President* instance);
};

CIMPLE_NAMESPACE_END

#endif /* _President_Provider_h */
```

C.2 President_Provider.cpp

```
#include "President_Provider.h"

CIMPLE_NAMESPACE_BEGIN

President_Provider::President_Provider()
{
}

President_Provider::~President_Provider()
```

```
{
}

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

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

Get_Instance_Status President_Provider::get_instance(
    const President* model,
    President*& instance)
{
    if (model->Number.value == 1)
    {
        instance = President::create(true);
        instance->Number.set(1);
        instance->First.set("George");
        instance->Last.set("Washington");
        return GET_INSTANCE_OK;
    }
    else if (model->Number.value == 2)
    {
        instance = President::create(true);
        instance->Number.set(2);
        instance->First.set("John");
        instance->Last.set("Adams");
        return GET_INSTANCE_OK;
    }
    else if (model->Number.value == 3)
    {
        instance = President::create(true);
        instance->Number.set(3);
        instance->First.set("Thomas");
```

```
        instance->Last.set("Jefferson");
        return GET_INSTANCE_OK;
    }

    return GET_INSTANCE_NOT_FOUND;
}

Enum_Instances_Status President_Provider::enum_instances(
    const President* model,
    Enum_Instances_Handler<President>* handler)
{
    President* instance;

    instance = President::create(true);
    instance->Number.set(1);
    instance->First.set("George");
    instance->Last.set("Washington");
    handler->handle(instance);

    instance = President::create(true);
    instance->Number.set(2);
    instance->First.set("John");
    instance->Last.set("Adams");
    handler->handle(instance);

    instance = President::create(true);
    instance->Number.set(3);
    instance->First.set("Thomas");
    instance->Last.set("Jefferson");
    handler->handle(instance);

    return ENUM_INSTANCES_OK;
}

Create_Instance_Status President_Provider::create_instance(
    President* instance)
{
    return CREATE_INSTANCE_UNSUPPORTED;
```

```
}

Delete_Instance_Status President_Provider::delete_instance(
    const President* instance)
{
    return DELETE_INSTANCE_UNSUPPORTED;
}

Modify_Instance_Status President_Provider::modify_instance(
    const President* model,
    const President* instance)
{
    return MODIFY_INSTANCE_UNSUPPORTED;
}

CIMPLE_NAMESPACE_END
```

Appendix D

The President Provider Registration Instances

This appendix defines the registration instances required to manually register the `President` provider.

```
instance of PG_ProviderModule
{
    Name = "Person_Module";
    Vendor = "Pegasus";
    Version = "2.5.0";
    InterfaceType = "C++Default";
    InterfaceVersion = "2.5.0";
    Location = "cimplePerson";
};

instance of PG_Provider
{
    Name = "Person_Provider";
    ProviderModuleName = "Person_Module";
};

instance of PG_ProviderCapabilities
{
    CapabilityID = "Person";
    ProviderModuleName = "Person_Module";
    ProviderName = "Person_Provider";
```

```
    ClassName = "Person";
    Namespaces = {"root/cimv2"};
    ProviderType = {2};
    supportedProperties = NULL;
    supportedMethods = NULL;
};
```

Appendix E

Document History

This appendix defines the history of this document. The version history of `CIMPLE` is defined on the `CIMPLE` web site.

Table E.1: **History**

Version	date	Description
1.0	2007	Original Release
2.0	March 2009	Update to CIMPLE Version 2. 1.Expand to include new chapters for logging, threading, upcalls, registration, wmi. 2. Included information on SFCB CIM Server usage. 3. Included informaton on WMI providers. 4. Add embedded instances documentation 5. Add new functions for list, scheduling 6. Correct minor editorial errors
2.1	September 2009	Expand detailed documentation. 1. Expand indication chapter (chapter 9) 2. Document condition variable broadcast method added in version 2.0.14 of CIMPLE. 3. Correct minor editorial errors
2.2	October 2009	Expand detailed documentation. 1. Correct minor editorial errors 2. Add index to this document 3. Add section defining CIMPLE Instance Map Class
2.3	September 2010	Expand detailed documentation. 1. Correct minor editorial errors 2. Expand index.

Index