Using CIMPLE

A Practical Guide to Developing CIM Providers

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

May 4, 2007

Using CIMPLE, Version 1.0

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 three prominent provider interfaces.

- OpenGroup CMPI Specification Version 2
- OpenPegasus C++ Provider Interface
- OpenWBEM C++ Provider Interface

With CIMPLE, developers can produce one provider that works with several CIM server implementations.

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 three different provider interfaces.

- OpenGroup CMPI Specification Version 2

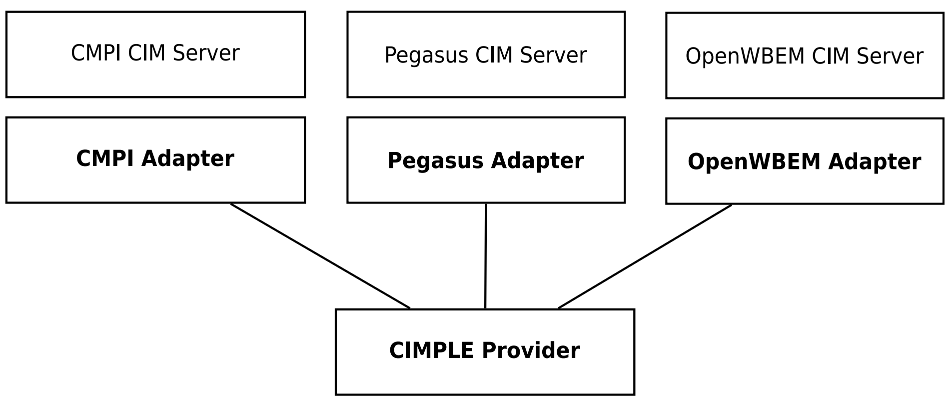
- OpenPegasus C++ Provider Interface
- OpenWBEM C++ Provider Interface

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 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.

```
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 OpenPegasus or CMPI is considerably more difficult. See section A.1 for the 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. This is needed in two situations.

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

In these situations, `genprov` patches the provider sources by rewriting the corresponding function signatures without disrupting anything else. `Genprov` patches the sources if they exist, else it creates them.

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, 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;
}
```

The following 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. 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 in terms of other operations.

- **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. For example, some instances providers only need to implement **enumerate-instances**.

Table 1.1: **Minimal Provider Impelementation**

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

1.4.5 Automated Provider Registration

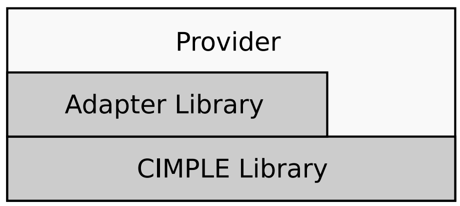
The CIMPLE `regmod` automates provider registration. For example, the following command registers all providers contained in `libPresident.so`.

```
$ 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` tool.

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, or OpenWBEM).



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

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 CMPI, OpenPegasus, and OpenWBEM. Please read section 2.2 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
- Windows IX86

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

2.1 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://cimple.org/downloads.html). Source distributions are available as gzipped tar files and as zip files. For example, the CIMPLE 1.0.0 source distribution is available as `cimple-1.0.0.tar.gz` and `cimple-1.0.0.zip`.

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

2.2 Configuring

To build CIMPLE standalone (without support for CMPI, OpenPegasus, and OpenWBEM), change to the CIMPLE root directory and type the following.

```
$ ./configure
```

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

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

To see how to configure CIMPLE for CMPI, OpenPegasus, or OpenWBEM, read the corresponding subsection below.

2.2.1 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.

2.2.2 Configuring for OpenPegasus – RPM Distribution

In general, you can configure for OpenPegasus with the following option.

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

`DIR` is the name of the directory where CIMPLE expects to find the OpenPegasus `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 OpenPegasus adapter, and the `regmod` tool.

2.2.3 Configuring for OpenPegasus – Source Distribution

To build CIMPLE for use with an OpenPegasus source distribution, use the following option in conjunction with the standard OpenPegasus environment variables.

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

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

```
PEGASUS_HOME
PEGASUS_ROOT
PEGASUS_PLATFORM
PEGASUS_DEBUG
```

This is equivalent to configuring with the following 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 `PEGASUS_HOME`.

2.2.4 Configuring for OpenWEBM

You can configure for OpenWEBM 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 Building

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

```
$ make
```

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

```
$ make check
```

This runs a handful of unit tests to see if the resulting build is usable on your platform.

2.4 Installing

To install CIMPLE, type the following command.

```
$ make install
```

This installs CIMPLE into the locations selected by the `configure` tool.

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. (`genclass`)
3. Generate the provider. (`genprov`)
4. Generate the module file. (`genmod`)
5. Implement the skeleton.
6. Build the provider.
7. Register the provider. (`regmod`)
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`.

```
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 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 3.6). The `-r` option creates `repository.h` and `repository.cpp`. Remember that since you will never edit the generated classes, you can regenerate them whenever you change the MOF definition.

When you develop a 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.

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 is a valid provider that provides zero instances of the `President` class. In section 3.5, 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.

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.

- CMPI
- OpenPegasus
- OpenWBEM

The following 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 3.6, we show how to compile `module.cpp` together with all the other generated files to build a module library.

3.5 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`

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

3.5.1 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`.

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

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.

- `ENUM_INSTANCES_OK`
- `ENUM_INSTANCES_FAILED`

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

3.5.2 Implementing the get_instance Stub

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

```
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. But providing a "proper" implementation of `get_instance` improves 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_INSTNACE_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.

- `ENUM_INSTANCES_OK`
- `ENUM_INSTANCES_FAILED`
- `ENUM_INSTANCES_UNSUPPORTED`

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

3.6 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.

3.6.1 Enabling a Provider Entry Point

The `module.cpp` file conditionally defines entry points for every supported provider interface (CMPI, OpenPegasus, OpenWBEM). 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)

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

```
-DCIMPLE_CMPI_MODULE
```

3.6.2 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
```

3.6.3 Linking the CIMPLE Libraries

The library (or DLL) must be linked with the correct CIMPLE libraries, which includes `cimple` and one of the following adapter libraries.

```
cimplecmpiadap (for CMPI)
cimplepegadap (for OpenPegasus)
cimpleowadap (for OpenWBEM)
```

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

3.6.4 Linking the Interface-Specific Libraries

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

Table 3.1: **Interface-Specific Libraries**

Provider Interface	**Required Libraries**
CMPI	
OpenPegasus	`pegprovider, pegcommon`
OpenWBEM	`owprovider, owcppprovifc, openwbem`

3.6.5 Position-Independent Code

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

3.7 Registering the Provider

This step is specific to OpenPegaus, which requires formal registration of providers. This involves creating registration instances in the OpenPegasus repository. Ordinarilly you do this by 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 `-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
```

3.8 Testing the Provider

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 3.7. 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.

3.8.1 Installing the Provider

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.

3.8.2 Enumerating Instance Names

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

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

3.8.3 Enumerating Instances

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";
};
```

3.8.4 Getting an Instance

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 CIM data types. Table 4.1 shows the correspondence between the two.

Table 4.1: **Data Types**

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. Arrays 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 defintion of the C++ `bool` type.

4.2 Integers

CIM 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**

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

Table 4.3: **Real Data Types**

CIMPLE Type Name	C++ Type
real32	float
real64	double

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 very basic `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.

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
```

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

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

```
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. 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());
}
```

Although `Array` is a template class, it does 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).

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 definitions, contained in `repository.mof`.

```
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 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.

```
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, `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 32 and generated on 33). 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 CIMPLE inheritance model, *static casting*, and *dynamic casting*. We preface our discussion with a cautionary note. Never use of 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 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

CIMPLE 1.0.0 does not support embedded instances; however, we are starting the implementation as we write this. Table 5.5 shows the status of embedded instances in recent versions of CIMPLE, Pegasus, and CIM.

To use embedded instances today, you need to use Pegasus 2.6.0 with an experimental version of CIM. We expect embedded instances to appear in the upcoming CIM 2.15 release. We expect CIMPLE embedded instances to be available in the next release (CIMPLE 1.0.1).

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.

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
Pegasus 2.5.2	no
Pegasus 2.5.3	no
Pegasus 2.5.4	no
Pegasus 2.6.0	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

```
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.

Instance Provider Methods
`load`
`unload`
`get_instance`
`enum_instances`
`create_instances`
`delete_instances`
`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 1.0.0 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;
}
```

The resource instance, discussed in the previous section, is constructed *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`. Else we return `MODIFY_INSTANCE_NOT_FOUND`.

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.

- Instance Providers
- Association Providers
- Indicaiton Providers

All three can 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 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-maker**. 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.
- A *logical return value* – the return value of the MOF method definition, returned in the `return_value` parameter.

The physical return value indicates whether the method is implemented or not (unsupported methods return `INVOKE_METHOD_UNSUPPORTED`).

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 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.

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 Pegasus `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 7.1 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 `enum_references_names`.

Chapter 9

Indication Providers

CIMPLE indication providers define the following 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.

We consider how to implement an indication provider that utilizes active generation. The provider creates a thread that publishes indications periodically.

9.1 The `OutOfOfficeNotice` Indication

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).

9.2 Implementing the `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.
    _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* used later 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 `_indicaiton_thread`, defined below.

```
void* OutOfOfficeNotice_Provider::_indication_thread(void* arg)
{
    OutOfOfficeNotice_Provider* provider =
        (OutOfOfficeNotice_Provider*)arg;

    while (provider->_continue.get())
    {
        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();

        Time::sleep(1 * Time::SEC);
    }

    return 0;
}
```

This function scan 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.

9.3 Implementing the `disable_indications` Method

The CIM server calls `disable_indication` when there are no longer any subscriptions to the `OutOfOfficeNotice` indication. Our implementation is shown below.

```
Disable_Indications_Status
OutOfOfficeNotice_Provider::disable_indications()
{
    // Destroy indication thread.
    _continue.dec();
    void* value_ptr;
    Thread::join(_thread, value_ptr);

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

    return DISABLE_INDICATIONS_OK;
}
```

This method performs the following steps.

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

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;
};
```