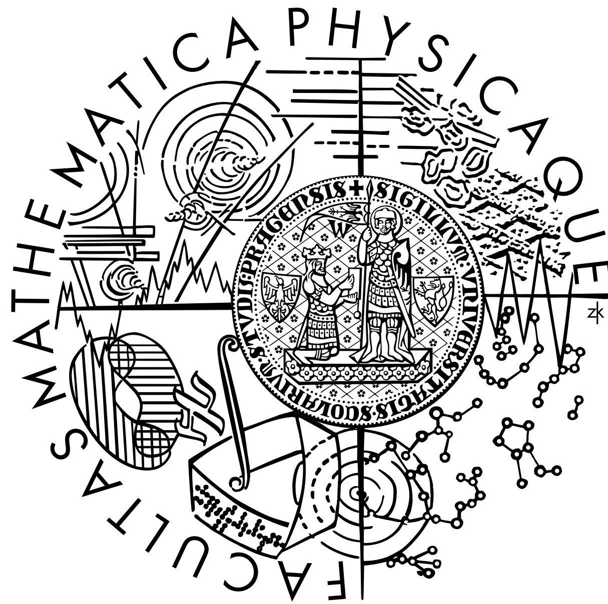
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

Diplomová práce



Martin Mrázik

Scripting of Common Information Model

Katedra softwarového inženýrství
Vedoucí diplomové práce: **RNDr. Stanislav Višňovský, Ph.D.**
Studijní program: **Informatika**

I would like to thank my supervisor, RNDr. Stanislav Višňovský, Ph.D., who made this thesis possible.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 10. 8. 2007 Martin Mrázik

Contents

1 Introduction **5**

1.1 Common Information Model 5

1.1.1 CIM Specification 7

1.1.2 CIM Schema 10

1.2 Web Based Enterprise Management 12

1.3 Goals 14

1.4 Structure of This Work 15

2 Scripting of CIM **17**

2.1 Related Works 17

2.1.1 Windows Management Instrumentation 17

2.1.2 pywbem 19

2.1.3 SBLIM 20

2.1.4 Other Projects 23

2.2 Scripting Language 24

2.3 Selected Python Features 25

3 Mapping CIM to Python **30**

3.1 Classes 30

3.2 Properties 34

3.3 Methods 35

3.4 Qualifiers 38

3.4.1 Generic Qualifiers 41

3.4.2 Qualifiers Specific for Classes 43

3.4.3 Property, Parameter and Method Qualifiers 45

3.5 Intrinsic Data Types 51

4 Implementation **52**

4.1 High-level Overview 52

4.1.1 WBEMConnection and its Subclasses 53

4.1.2 CIMClass and ResultContainer 55

4.1.3 WBEMFactory and ClassFactory 57

4.1.4 Qualifier and Property 61

4.1.5 SelfReprType and EncapsulateCIMErrorType 61
4.2 Extrinsic Methods 62
4.3 Logging and Debugging 63
4.4 API Documentation 64
4.5 Unit Tests 64
4.6 Integration With IPython 65

5 Summary **67**
5.1 Open Issues and Future Work 69

A PowerCIM Examples **71**
A.1 Start/Stop Services – Thick Client 71
A.2 AppArmor Security Event Notification 74

B Default Logging Configuration **78**

C IPython configuration **80**
C.1 ipythonrc 80
C.2 powerCIM-magic.py 81

D PowerCIM DVD **82**

Title: Scripting of Common Information Model

Author: Martin Mrázik

Department: Department of Software Engineering

Supervisor: RNDr. Stanislav Višňovský, Ph.D.

Supervisor's e-mail address: `visnov@suse.cz`

Abstract: This work investigates how system management applications could be rapidly developed with management protocols defined by Distributed Management Task Force (DMTF) and open-source programming languages. The focus is on Common Information Model (CIM), Web Based Enterprise Management (WBEM) and scripting languages.

CIM and WBEM are briefly described and the basic terminology of these standards and their infrastructure is introduced. Goals such as object-oriented mapping, shell enablement or generating documentation for CIM classes are discussed.

Similar work in this area with their strengths and weaknesses are covered. Requirements for a scripting language are stated and Python is selected for this work with a discussion of some of its more interesting features. Later, Python mapping of Common Information Model is defined with the focus on Python features and conventions.

As a proof of concept a prototype mapping (called powerCIM) is implemented and a few examples of management scripts are provided demonstrating the synergy with other open-source projects and libraries such as IPython or PyQt.

Keywords: System Management, WBEM, CIM, Python.

Název práce: Scripting of Common Information Model

Autor: Martin Mrázik

Katedra (ústav): Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Stanislav Višňovský, Ph.D.

e-mail vedoucího: `visnov@suse.cz`

Abstrakt: Tato práce zkoumá možnosti rychlého vývoje aplikací použitím protokolů definovaných sdružením "Distributed Management Task Force" a použitím programovacích jazyků ze světa otevřeného software. Je zaměřena hlavně na protokoly "Common Information Model" (CIM), "Web Based Enterprise Management" (WBEM) a skriptovací programovací jazyky.

Standardy CIM a WBEM jsou stručně popsány a čtenář je seznámen se základní terminologií těchto standardů a s jejich infrastrukturou. Jsou definovány cíle jako objektově-orientované mapování, generování dokumentace nebo podpora pro interaktivní konzoli.

Probrány jsou související práce v dané oblasti s rozborem jejich silných a slabých stránek. Jsou definovány požadavky na skriptovací jazyk. Je vybrán jazyk Python a jsou uvedeny některé jeho zajímavé vlastnosti. Později je definováno mapování standardu "Common Information Model" do jazyka Python se zaměřením na jeho vlastnosti a konvence.

V jazyku Python je naimplementováno prototypové řešení (nazvané powerCIM) spolu s několika ukázkami skriptů, ktoré zároveň demonstrují synergický efekt této práce a dalších projektů ze světa otevřeného software jako například IPython nebo PyQt.

Klíčová slova: System Management, WBEM, CIM, Python.

Chapter 1

Introduction

The complexity of large, geographically distributed, heterogeneous network (different architectures and operating systems, wide range of peripherals) poses many challenges for businesses relying on this infrastructure. In such computing environment a lot of resources are required for maintenance, upgrades, patching, system monitoring and similar tasks. To make these tasks easier industry standards such as Simple Network Management Protocol (SNMP) or Common Information Model (CIM) has arisen, addressing this specific problem.

In the late 1990s Distributed Management Task Force[1] defined two closely related standards – Common Information Model (CIM) and Web Based Enterprise Management (WBEM). CIM/WBEM is currently being adopted by many enterprises and its importance is growing. This work is focused on this set of standards.

1.1 Common Information Model

Common Information Model provides object oriented tool-set to model entities being managed (e.g. network services, applications, servers etc). This tool-set is based on Unified Modeling Language[2] (UML) and thus the basic constructs will be familiar to many developers (according to *Methods & Tools* around 50% of software development organizations has totally or partially adopted the UML techniques [37]). CIM also provides a modeling language based on Interface Definition Language (IDL) called Managed Object Format (MOF) which will be discussed in Section 1.1.1.

Suppose we have two physical servers (A, B) both running different operating systems. These operating systems provide virtualization services and both are running several virtual machines on top of them. CIM enables us to model

[1]http://www.dmtf.org
[2]http://www.uml.org

these entities (also called *managed elements*), their services and relationships in a hierarchical fashion.

In this example we want to model the following managed elements:

- generic operating system
- hosting operating system with virtualization capabilities (running on server A and B)

In CIM, managed elements are modeled as classes and it is possible to use standard OOP constructs such as inheritance, for modeling the systems. In this example "generic operating system" will serve as a base-class for more specific operating system – an operating system that provides virtualization services.

Operating systems may provide the following services in our model:

- restart the system
- shutdown the system
- get some statistics such as memory/disk space usage, CPU utilization, etc

Hosting operating system could be able to do also the following:

- enumerate all virtual systems running on top of this system
- start a virtual system (this includes configuration of the virtual system, i.e. which virtual devices I need, how much memory, how many virtual CPUs etc)
- stop a virtual system
- make a snapshot of virtual system

Using the tools provided by CIM it is possible to express classes (operating system), instances (operating system A), methods (restart system), associations between them (virtual system X is running on top of operating system A) and so on.

CIM is comprised of two parts – the specification (defining basic modeling constructs) and the schema (set of classes which were identified as most common building blocks for other classes).

CIM is focused only on the modeling of managed elements and their relationships. It does not provide a protocol how to access this management functionality. The communication problem is solved by another DMTF's standard – Web Based Enterprise Management[3] (WBEM), which will be discussed later in Section 1.2.

[3]http://www.dmtf.org/standards/wbem

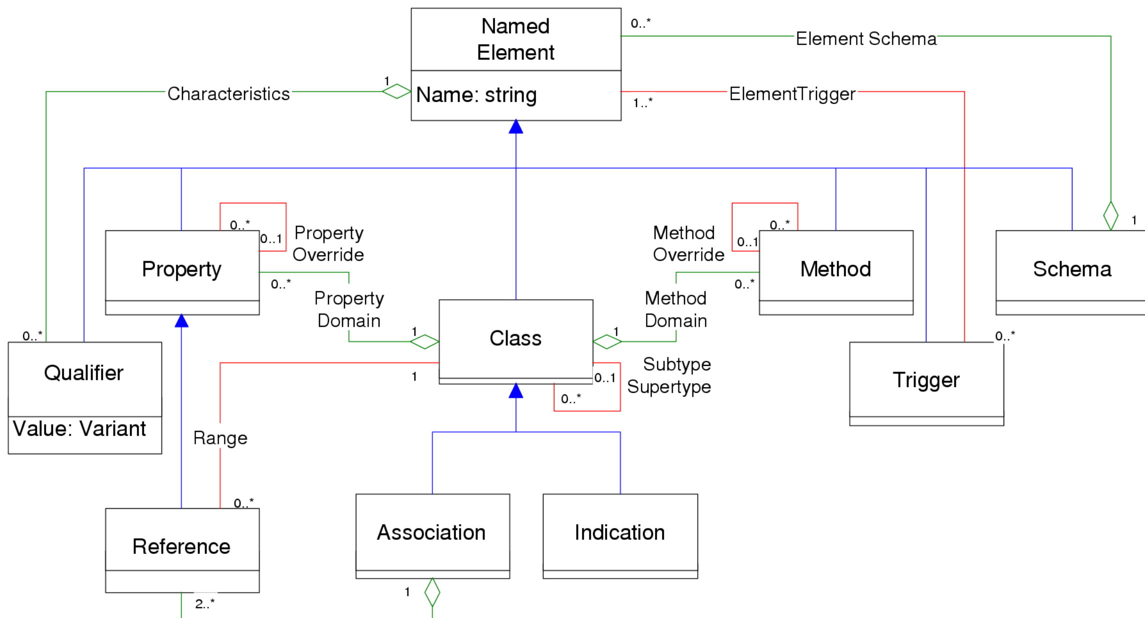


Figure 1.1: Meta Schema Structure [4]

1.1.1 CIM Specification

CIM Specification provides Meta Schema which is a formal definition of the model. Meta Schema defines basic building blocks (such as `class`) that can be used to describe the model and its usage and semantics. Figure 1.1 provides an overview of the Meta Schema structure using Unified Modeling Language. The elements of a model are[4]:

Schema is a group of classes with single owner. It could be understood as a namespace. Each class must have an unique name within the schema it is defined in.

Class, Method and Property have their traditional meaning from object oriented modeling. Classes may participate in Associations.

Trigger is recognition of a state change (such as create, delete, update or access) of Class instance and update or access of a Property [4].

Indication is an object created as a result of trigger. Indication object is a representation of an event.

Association is a representation of a relationship between two classes. Since associations are not an integral part of a class, the relationship does not

[4]Please note that this description is just an introduction. Full specification can be found in [4].

affect the interface of a class. Only association classes can have references to other classes.

Reference is a pointer to other class instance.

Qualifier is used to characterize named elements. Qualifiers also provide a mechanism how to make the meta schema extensible. For example using a qualifier it is possible to define that a given class is abstract or a given property has some min/max constraints (e.g. an integer property must be greater than 100 but less than 200).

The Meta Schema defines also basic data types (such as `uint8`, `boolean`, `real32`, `datetime`, etc) and a set of common qualifiers (e.g. `Description` – textual description of a class, property or method; `Maxvalue` – maximal value of a property, parameter or return value of a method, etc).

To express both the Meta Schema and custom schemas, CIM Specification defines a language based on Interface Definition Language (IDL) called Managed Object Format (MOF). This language, together with UML, is the primary tool for describing custom models. Formal definition of MOF grammar and meta schema can be found in [4].

CIM Specification defines also a naming mechanism for classes. A fully qualified class name is in the form <schema_name>_<class_name>.

With CIM Specification we can formally model our example with operating systems and virtualization services.

In Figure 1.2 and Figure 1.3 you can see an example how different meta schema constructs could be used to describe custom classes (managed elements) in MOF and UML language respectively. Both figures are modeling the same example – an operating system and operating system with virtualization capabilities as discussed in Section 1.1.

In this example `ACME_OperatingSystem` (`OperatingSystem` class in an `ACME` schema) is a subclass of `ACME_ManagedElement`. The class has several properties (e.g. `Name` – string property; `FreePhysicalMemory` – unsigned 64bit integer) and two methods (`Shutdown` and `Reboot`).

Each property has a `Description` qualifier with a string value – it contains a textual description of the property. An important property qualifier is `Key` (line 4 in Figure 1.2). Set of `Key` properties uniquely identifies an instance of a given class (a good analogy to key/class in CIM specification is a primary key/table in relational databases).

The `Shutdown` method (Figure 1.2, lines 17-26) has one input (`IN` qualifier is set to `True`) parameter – `TimeInterval`. It defines how many seconds should the system wait before performing the actual shutdown. `Units` is a string qualifier defining the units of the input parameter (in this case it is seconds).

`ACME_HostingOperatingSystem` is an OperatingSystem that provides virtualization services. In this example only `StartVirtualSystem` method is defined

```
class ACME_OperatingSystem : ACME_ManagedElement {
        [ Description (
            "Fully Qualified Domain name identifying this operating system"),
          Key(True)]
    string Name;

        [ Description ("A string indicating the type of operating system.")]
    string OSType;

        [ Description ("Number of Kbytes of physical memory currently unused."),
          Units("KiloBytes")]
    uint64 FreePhysicalMemory;

        [ Description ("Reboot the operating system.")]
    uint32 Reboot();

        [ Description ("Shutdown the operating system.")]
    uint32 Shutdown(

          [ IN(True),
            Units("Seconds"),
            Description (
              "Wait TimeInterval seconds before starting the shutdown "
              "process. A value of 0 indicates that the system "
              "should be shutdown immediately.")]
        uint32  TimeInterval);
};

class ACME_HostingOperatingSystem : ACME_OperatingSystem {
    [ Description("Define and start a virtual system.")]
    uint32 StartVirtualSystem(
          [ IN, Description("Number of virtual CPUs for this system.")]
        uint32 NumberOfCPUs,
          [ IN, Description("Hard-disk size of this virtual system."),
            Units("Bytes")]
        uint32 DiskSize,
          [ IN, Description("Size of RAM."), Units("Bytes")]
        uint32 RAMSize,
          [ IN, Description(
            "Boot device. Can be either path to a device or iso image.")]
        string BootDevice)
/* ... */
};
```

Figure 1.2: Operating System Example (MOF)

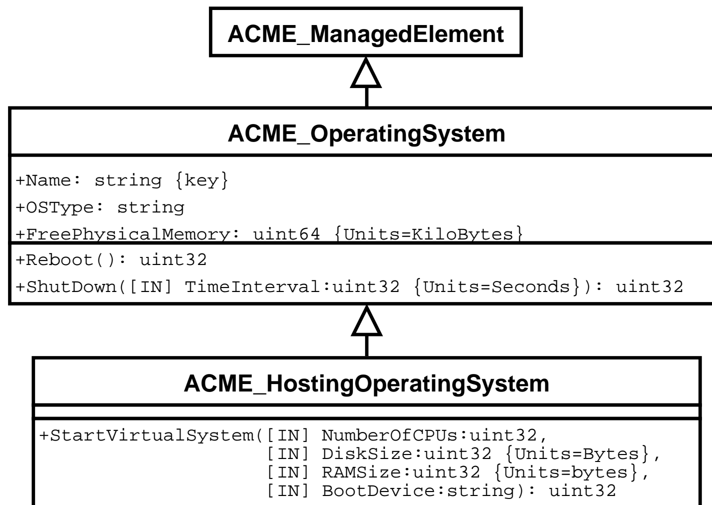


Figure 1.3: Operating System Example (UML)

for simplicity. Other methods (such as stop system, make a snapshot) could be added in similar way. `StartVirtualSystem` starts a virtual system with given number of CPUs, given disk/RAM size and a defined boot device.

This example is supposed to give an overview how a CIM model looks and how meta schema constructs could be used. It is far from being useful for real management of virtual systems. To make it little bit more useful an association could be defined to capture the relationship between different operating systems (hosting and virtualized).

1.1.2 CIM Schema

Having an object-oriented modeling framework forces the designer to create a hierarchical model where most simple classes are at the top of the inheritance tree. This is one of the basic concepts of object-oriented modeling which makes modeling of complex structures easier.

CIM Schema defines a basic set of classes divided into three categories that could be extended to meet specific needs of a designer. These categories are Core Model, Common Model and Extension Schema.

Core Model

The Core model is the smallest set of classes, associations and properties providing basic elements for building managed systems. This is the starting point for designer who is designing any managed system. Although some classes might be added to Core Model in the future, any major changes in existing classes are very unlikely [4]. The Core Model includes classes like `CIM_LogicalDevice` or

CIM_PhysicalElement, i.e. abstract[5] elements.

Common Model

The Common Model is basic set of classes that define various technology independent areas [4]. Classes (methods, associations, ...) provided in Common Model are supposed to be detailed enough to serve as a basis for program design and, in same cases, implementation.

The common models for CIM Schema are divided into following categories:

Application models provide abstractions needed for deployment, monitoring and maintenance of software products. For example CIM_J2eeEJB and CIM_InstalledProduct classes are defined here.

Event models represent events (e.g. change in the state of environment) and alarms. Different classes are provided such as CIM_ClassIndication or CIM_InstCreation.

Network models describe various network devices, protocols and services (e.g. CIM_SwitchPort).

Support models describes "object and transaction models for the exchange of knowledge related to support activities (Solutions) and the processing of Service Incidents" [7]. Example classes are PRS_Solution and PRS_Activity.

Database defines models for database environment such as CIM_DatabaseService or CIM_SqlTable.

Interop model describes the WBEM infrastructure and how other components (CIM Clients, Managed Elements, ...) interact with this infrastructure. It defines classes such as CIM_WBEMService or CIM_Error.

Physical model defines inventory and asset management related classes such as CIM_Magazine or CIM_Container.

Systems models describe computer-system related abstractions. They abstract aggregation of "parts" that form a single, manageable entity. Classes provided here are for example CIM_ComputerSystem or CIM_FileStorage.

Devices models describe hardware functionality, their configuration and state information (e.g.CIM_Printer, CIM_TapeDrive).

Metrics model defines "the management components that allow the dynamic definition and retrieval of metric information" [15]. It defines classes such as CIM_UnitOfWork.

[5] In this context abstract means "existing as an idea, feeling or quality, not as a material object" [3].

Policy model provides rules and associated actions such as `CIM_Authorization-Rule` or `CIM_BiometricAuthentication`.

User provides abstraction for location, identity and authority of a user (e.g. `CIM_User`, `CIM_OrgUnit`).

Extension Schema

The extensions schemas are technology-specific extensions to the Common model. It is a place where vendor specific schemas belongs. It is also expected that some models from Extension Schema might be "promoted" to Common model as they become widely adopted.

`Xen_VirtualSystemManagementService` is an example of a model in Extension Schema. It models a service manipulating XEN[6] virtual systems and their components.

1.2 Web Based Enterprise Management

Another standard closely related to CIM is Web Based Enterprise Management (WBEM)[7]. While CIM is focused on the modeling part, WBEM provides a set of standards defining how to access the entities modeled in CIM and what operations are available. WBEM addresses the problem of communication between *operators* (those who are operating the modeled environment) and *managed elements*.

WBEM defines xmlCIM [6] – an encoding specification. It defines a DTD[8] grammar which can be used not only to represent CIM entities (classes, properties, etc) in XML[9] but also to express CIM messages (e.g. calling a method of a class, getting a class instance, etc). By mapping CIM constructs and messages into XML document it is possible to transport them using a standard protocol such as HTTP.

Although it is possible to map CIM to XML using xmlCIM, the operations with a model and/or schema are still not defined. To fill this gap two standards exists at the moment – CIM-XML and WS-Management. In the future other standard could be added (currently WSDM is under discussion).

CIM-XML (defined in [5]) is a protocol that uses XML (xmlCIM) over HTTP to exchange CIM information. It defines a CIM operation requests which are invocations of one or more methods. Method could be either *intrinsic* (operation that enables to work with CIM object model) or *extrinsic* (defined as a

[6] Xen is a software virtual machine monitor. It allows to run several guest operating systems on the same hardware at the same time. It is available from http://www.xensource.com/ [14].

[7] http://www.dmtf.org/standards/wbem

[8] Document Type Definition (DTD) is discussed in http://www.w3.org/TR/2000/REC-xml-20001006#dt-doctype

[9] Extensible Markup Language (http://www.w3.org/XML/)

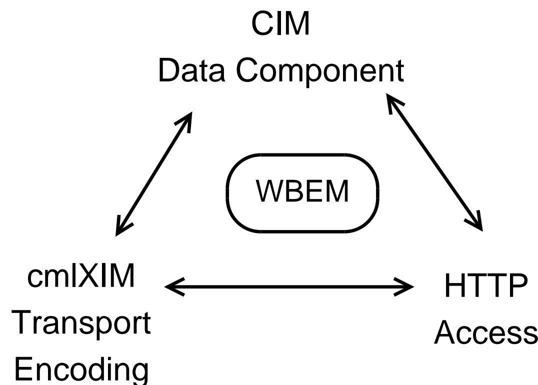


Figure 1.4: WBEM Components [1]

method on a CIM class or schema – e.g. `Reboot` in `ACME_OperatingSystem` class). Intrinsic methods are for example `GetInstance`, `EnumerateClasses` and `ModifyInstance`. Figure 1.4 displays how all these WBEM components fit together.

WS-Management (Web Services for Management [9]) is an alternative standard to CIM-XML. It uses Simple Object Access Protocol (SOAP) to achieve the same as CIM-XML. However, this standard is still in its preliminary release.

The overall architecture of CIM/WBEM enabled infrastructure is illustrated in Figure 1.5. In this figure, *operator* is represented by `WBEM Client` and `WBEM Listener`[10].

The *managed elements* in Figure 1.5 are `Services`, `Hardware` and `Software`. These elements provide management functionality that should accessible via WBEM (e.g. start or stop a service).

WBEM Server closes the "management gap" between *operators* and the *managed elements* (hardware and/or software components). It stores the model (CIM classes) in a `Repository`.

To separate specific knowledge of managed elements (e.g. how to find out free disk space on a given file system) from WBEM Server, so called providers exists. Providers know the internals of a managed element and they implement the model using this knowledge. The functionality is then provided to WBEM server (providers can be thought as a drivers for the managed element) and thus the primary role of a WBEM server is to act as a broker between CIM clients and providers.

[10]WBEM Listener ("Event Operator's Workstation") is the place where indications (see Section 1.1.1) are sent from WBEM Server.

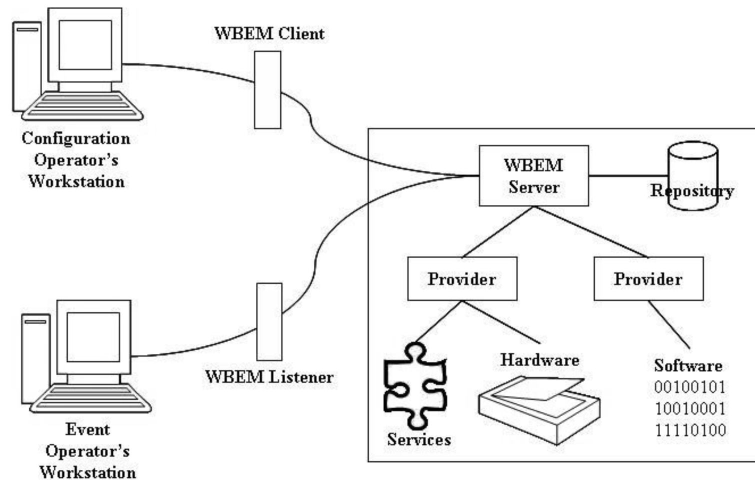


Figure 1.5: WBEM Server, Clients and Providers [1]

1.3 Goals

This work is focused only on the client's part of Common Information Model. It should provide a mapping of Common Information Model to a selected scripting language (choosing a suitable language is part of this work) and a prototype implementation of the mapping. The prototype should allow the developers to write applications or helper scripts that can access the WBEM server and perform management operations.

The mapping is supposed to hide as much technical information about CIM as possible and provide an environment which programmers are familiar with. Writing CIM clients quickly and efficiently is the common denominator for most of the goals discussed in this section.

Specifically these areas will be addressed:

Object-Oriented Programming. At places where it makes sense, object oriented practices should be applied. Common Information Model itself is object oriented, therefore this is the most natural approach. Most of the programmers are familiar with OOP and therefore it will be easier to use the library for them.

Probably the most important is mapping from the CIM Model to the selected programming language. There should be a clear relationship between CIM constructs (classes, methods, instances, qualifiers, etc) and constructs natively used in the selected programming language (e.g. in C++, CIM classes would be mapped directly to C++ classes as opposed to mapping to Document Object Model[11] structure obtained directly from xmlCIM which would be also possible).

[11] http://www.w3.org/DOM/

All other constructs (inheritance, virtual methods, etc) should be naturally presented to the programmer where it makes sense (e.g. most scripting languages will not be able to express "Units" qualifier for a class property[12]).

Concepts widely used in the selected scripting language should be adopted in the library not to confuse programmers already familiar with them (e.g. in Python each CIM class/method should have the "__doc__" docstring with documentation string [29]) and to make integrating with other standard libraries easier.

Documentation. The library itself must be well documented so the programmers can start writing CIM clients as fast as possible. It should also provide ways how to generate documentation for the CIM model (i.e. for CIM classes, methods, etc) in a way natural for the selected programming language (e.g. in Java it should be possible to generate Javadoc[13] documentation for the CIM classes).

Debugging. No software is bug-free, thus making debugging of the client code as easy as possible will make the development faster. Because remote objects will be accessed (managed elements are most likely to be located at different place than the operator/CIM Client), debugging support is even more important.

Use of Scripting Language. This is closely related with the "Shell Enablement" goal (see below). The selected scripting language should be widely used by the community of developers. According to some views, development in scripting languages is also more productive [23].

"Openness". The solution should be available on different platforms (Linux, Windows, ...) to reach a broad audience of developers. Proprietary solution would be a disadvantage. This work was inspired by a lack of suitable tools on Linux platform.

"Shell Enablement". Optionally it would be good to have an easy-to-use shell environment which could be used as a generic console for accessing a WBEM Server. This is similar to Windows PowerShell (codename Monad)[14].

1.4 Structure of This Work

Chapter 2 discusses related work, why it is important to choose the scripting language early in the design stage and the choice of a suitable scripting language.

[12] "Units" is a string qualifier which indicates the units of the associated data item (e.g. "KiloBytes").

[13] http://java.sun.com/j2se/javadoc/

[14] http://www.microsoft.com/windowsserver2003/technologies/management/powershell/default.mspx

It also introduces some of the features of the selected language and how these features can be used.

Chapter 3 discusses the mapping of CIM and its constructs to Python so the goals of this work are fulfilled. If there are more options how to map the same construct, they are discussed with their advantages and disadvantages.

Chapter 4 describes how the prototype solution is implemented providing a high-level UML overview. Some of the technical nuances are discussed in this chapter. It also provides an overview of the API available to the CIM client developer and details about integration with other tools (namely with IPython).

Chapter 5 provides overview of what was achieved, how this work compares to related work in this field and what are the open issues and future work.

Appendix A provides a few examples how the prototype implementation could be used to write management applications. These examples are accompanied with a short discussion.

Appendix B contains the default configuration of logging used by the prototype implementation. This configuration file is specific for `logging` module provided by the standard Python library.

Appendix C provides details and configuration files for integrating this work with IPython.

Appendix D describes how to use the live DVD that comes with this work. The prototype implementation of CIM to Python mapping is preinstalled and it is possible to use this DVD as a demonstration of powerCIM capabilities. This DVD also contains source code of this prototype and a PDF version of this work.

Chapter 2

Scripting of CIM

2.1 Related Works

This work is definitely not the first one trying to solve the problem of developing CIM clients. However, most of the existing projects do not fulfill all the goals defined in Section 1.3. In many cases they are very difficult to use without a deeper knowledge of Common Information Model and all related standards.

This section summarizes some publicly available CIM scripting projects and for each project a simple example demonstrating its capabilities is implemented. This example will print all running services, their status and will stop each service on `localhost`. It will demonstrate the most common usage of CIM/WBEM – getting an instance, accessing properties and calling instance methods.

2.1.1 Windows Management Instrumentation

Windows Management Instrumentation (WMI) is implementation of the WBEM and CIM standards by Microsoft. Not only comes WMI pre-installed on different versions of Windows operating systems (such as Windows Me, Windows XP, etc [13]) but it is also possible to manage them using scripting tools provided by Microsoft.

Microsoft provides three options for WMI scripting – VBScript, Windows PowerShell and WMIC (Windows Management Instrumentation Command-line).

WMIC is a command-line tool and its main purpose is to ease WMI information retrieval. WMIC is not supposed to be used for development of sophisticated scripts.

On the other hand, VBScript and Windows PowerShell are multi-purpose scripting projects with the ability to directly operate on WMI. In some concepts they are very similar – both are object-oriented and both can be used for development of complex scripts. While VBScript (Visual Basic Scripting Edition) is based on Visual Basic, Windows PowerShell uses some of the C# syntax.

```
Set objWMIService = GetObject("winmgmts:\\localhost\root\cimv2")
Set colItems = objWMIService.ExecQuery("Select * From CIM_Service")

For Each objItem in colItems
    Wscript.Echo objItem.Name, objItem.Started
    Return = objItem.StopService()
Next
```

Figure 2.1: Example – List Services Using Microsoft VBScript [27]

```
$colItems = get-wmiobject -class "CIM_Service" `
-namespace "root\cimv2" -computername "localhost"

foreach ($objItem in $colItems) {
      write-host $objItem.Name, $objItem.Started
      $Return = $objItem.StopService()
}
```

Figure 2.2: Example – List Services Using Windows PowerShell [27]

You can find the "list and stop all services" example for VBScript and Windows PowerShell in Figure 2.1 and Figure 2.2 respectively. In both examples code on lines 1-2 enumerates all instances of `CIM_Service` class running on server "`localhost`". The rest of the code is straightforward in both cases – lines 4-7 iterates over all services. For each service its name and state is printed and eventually the service is stopped. Return value of "`StopService`" method is stored in "`Return`" variable. As you can see although the syntax is different, the structure of the code is very similar.

These projects are well documented on Microsoft web sites. Many articles, examples and blogs discussing them exists, what creates an outstanding knowledgebase for further development. However, the biggest disadvantage is that all these tools are Windows centric. For example, it is not possible to deploy scripts written in Windows PowerShell to operator running thin-client terminal based on Linux. It is impossible to develop CIM clients that will run on different operating systems as well.

Another products exists for Windows as well (e.g. WMI for Python[1]) but they do not provide any major advantage comparing with the products shipped by Microsoft.

[1] This module requires pywin32 extension and thus it is only relevant for Windows although Python itself is a multi-platform scripting language. WMI extension for Python could be downloaded from http://cheeseshop.Python.org/pypi/WMI/1.3

```
import pywbem

conn = pywbem.WBEMConnection('http://localhost', ('', ''), 'root/cimv2')
instanceNames = conn.EnumerateInstanceNames('CIM_Service')

for instanceName in instanceNames:
    instance = conn.GetInstance(instanceName, LocalOnly=False)
    name = instance.properties['Name'].value
    started = instance.properties['Started'].value
    print name, started
    Return = conn.methodcall('StopService', instanceName)
```

Figure 2.3: Example – List Services Using pywbem

2.1.2 pywbem

Pywbem[2] is a Python module implementing the CIM-XML standard and providing simple mapping of CIM elements (classes, qualifiers, etc) to Python objects. It enables the developer to write Python scripts that can access a WBEM server via CIM-XML and execute CIM operations defined in [5].

Although pywbem provides mapping between Python and CIM constructs it is not very straightforward. For example it provides `CIMInstance` object that maps CIM instances to Python. However all properties and methods are stored in a special dictionary (associative array) attribute. This makes developing CIM scripts more difficult, time consuming and less intuitive.

In Figure 2.3 you can see a code snippet that lists all services, their status and stops them. Line 3 defines a connection to WBEM server in "`root/cimv2`" namespace. Next line enumerates all instance names (instance name is a class name and values of key properties – instance name uniquely identifies an instance). In this case the example does not enumerate instances directly, because calling a method requires instance name as a parameter[3].

Line 7 gets the `CIM_Service` instance (`LocalOnly=False` parameter specifies that attributes inherited from base class should be fetched from the server as well) while line 10 prints the name and status of this service. Finally, line 11 calls the `StopService` method which will stop the service.

This code is much more difficult to write and understand than the example written in VBScript or Windows PowerShell. It requires non-trivial distinction between instance and instance name and access to special dictionary properties (lines 8 and 9).

[2] http://pywbem.sourceforge.net/

[3] Although it would be possible to get instance name from instance it would require to process all properties and find out which of them are "key".

```
1  wbemcli ein http://localhost:5988/root/cimv2:CIM_Service |
2  while read instanceName; do
3
4      instance=`wbemcli gi http://$instanceName`
5      Name=`echo $instance | sed -n 's/^.*,Name="\([^"]*\)",.*$/\1/p'`
6      Started=`echo $instance | sed -n 's/^.*,Started=\([^,]*\),.*$/\1/p'`
7      echo "$Name $Started"
8
9      Return=`wbemcli cm http://$instanceName StopService`
10 done
```

Figure 2.4: Example – List Services Using wbemcli/bash

The last problem of pywbem is its documentation. The whole documentation consists only of few examples. This implies some reading of pywbem code during development of custom CIM clients. Although the structure of pywbem is clean and simple it makes the development little bit more demanding.

Unfortunately pywbem does not provide support for WS-Management as additional WBEM protocol.

2.1.3 SBLIM

Standards Based Linux Instrumentation for Manageability (SBLIM)[4] is a project that provides open source implementation of WBEM-based solutions for Linux. It provides projects such as Small Footprint CIM Client (library that enables the developer to write CIM clients in C), CIM Client for Java (library for writing CIM clients in Java), tools for creating CIM models in UML (ECUTE – Extensible CIM & UML Tooling Environment) and many others. SBLIM is driven by WBEMSource – an initiative promoting the widespread use of management technologies defined by DMTF.

This section will discuss only CIM Client for Java and SBLIM wbemcli[5].

SBLIM wbemcli

SBLIM wbemcli is a command line utility which could be used directly from command line or together with some unix shell (e.g. bash) to write simple management scripts.

[4]http://sblim.wiki.sourceforge.net/

[5]Although Small Footprint CIM Client also enables the developers to write CIM clients, it is using an approach very similar to CIM Client for Java and therefore will not be discussed in this section.

```
king:~ # wbemcli ein http://localhost:5988/root/cimv2:CIM_Service
localhost:5988/root/cimv2:OMC_ServiceFromXML.SystemCreationClassNa ↩
me="OMC_UnitaryComputerSystem",SystemName="king.suse.cz",CreationC ↩
lassName="OMC_ServiceFromXML",Name="apache2"
localhost:5988/root/cimv2:OMC_ServiceFromXML.SystemCreationClassNa ↩
me="OMC_UnitaryComputerSystem",SystemName="king.suse.cz",CreationC ↩
lassName="OMC_ServiceFromXML",Name="postfix"
localhost:5988/root/cimv2:OMC_ServiceFromXML.SystemCreationClassNa ↩
me="OMC_UnitaryComputerSystem",SystemName="king.suse.cz",CreationC ↩
lassName="OMC_ServiceFromXML",Name="dhcpd"
```

Figure 2.5: Wbemcli – Enumerate Instance Names

From the command-line nature and the fact that bash has no native support for objects it is clear that using wbemcli for more sophisticated scripting would be very difficult.

In Figure 2.4 wbemcli is used to enumerate all instance names (`ein`) of `CIM_Service` class on line 1. The same problem discussed in pywbem example (Section 2.1.2) with instances and instance names applies for wbemcli as well. It is not possible to enumerate instances directly, because calling a method requires an instance name.

"`wbemcli ein`" command prints each instance to standard output on a single line. Properties of an instance are delimited by a comma (",") character (you can see an example of `ein` command output in Figure 2.5). Each line with an instance is read and processed further with standard unix tools such as `sed` or `read`. `StopService` method is called (`cm`) on line 8, storing the result in `Return` variable.

Code like this is very difficult both to develop and maintain (and this snippet is not even bug-free – consider the fact that the `Name` attribute might contain double quotes (") character).

CIM Client for Java

The SBLIM CIM Client for Java is an implementation of WBEM services client. It is a java library that allows the users to develop CIM clients (management applications) in java.

Figure 2.6 shows the "list services" example implemented in java using this library. Lines 9-11 define a connection to the WBEM server (`localhost`). Line 14 enumerates all instances of the `CIM_Service` class. For each instance (line 15) its name and status is printed (lines 19-21). Finally the `StopService` method is called on lines 24 and 25.

```java
import org.sblim.wbem.client.*;
import org.sblim.wbem.cim.*;
import java.util.Vector;
import java.util.Enumeration;

class ListServices {
    public static void main(String[] args) {

        CIMClient cimClient = new CIMClient(
                new CIMNameSpace("http://localhost", "root/cimv2"),
                new UserPrincipal(""), new PasswordCredential(""));

        CIMObjectPath op = new CIMObjectPath("CIM_Service");
        Enumeration instances = cimClient.enumerateInstances(op);

        while (instances.hasMoreElements()) {
            CIMInstance service = (CIMInstance) instances.nextElement();
            CIMProperty started = service.getProperty("Started");
            System.out.print(service.getProperty("Name").getValue().toString()
              + " " +
              service.getProperty("Started").getValue().toString()
            );

            CIMValue Return = cimClient.invokeMethod(service.getObjectPath(),
                    "StopService", new Vector(), new Vector());
        }
    }
}
```

Figure 2.6: Example – List Services Using CIM Client for Java

2.1.4 Other Projects

Besides the projects mentioned in previous sections there are others that are trying to make the development of WBEM/CIM components easier, or they can help with fulfilling the goals as defined in Section 1.3. However, they are either beyond the scope of this work or have similar problems as projects discussed earlier.

rubywbem is a port of pywbem to Ruby programming language[6] with all the disadvantages discussed in Section 2.1.2 (actually there are more – there is no documentation for rubywbem at all and it seems this project is not actively developed).

IPython is a project that provides enhanced shell for Python[7]. It has many nice features for object introspection (e.g. it is possible to access docstrings of Python objects using a special, very terse, syntax), it provides extensible "magic" commands (used e.g. to execute shell commands), complete shell access and many, many others.

It can be used for Python development (it provides enhanced debugging support) and as a generic purpose system shell. Please consult [21] for more information about IPython and its features.

pydoc, epydoc are two different tools for generating API documentation from Python source code. While pydoc is part of standard Python library, epydoc[8] is an independent project trying to create a sophisticated tool with more features than pydoc and output that resembles what doxygen[9] or javadoc[10] produces.

WBEM Services is an open-source project developing Java implementation of WBEM/CIM. It provides WBEM server, Java APIs, MOF to JavaBeans generator and other sub-projects.

openWBEM/openPegasus are open-source implementations of CIM/WBEM standards. Both projects are focused mainly on the server side of the problem – i.e. they solve how to write providers (using different languages such as C++ or Perl) and both are implementing WBEM server (CIM operations over HTTP as defined in [5]). OpenPegasus provides a command line utility for accessing the WBEM server but this utility is even more difficult to use than wbemcli, because the user is forced to encode their operations manually to xmlCIM.

[6]http://www.ruby-lang.org/en/
[7]http://ipython.scipy.org/
[8]http://epydoc.sourceforge.net/
[9]http://www.stack.nl/~dimitri/doxygen/
[10]http://java.sun.com/j2se/javadoc/

Openwsman is an open-source implementation of the Web Services Management (WS-Management) specification. It provides WS-Management server, command line utility (wsmancli), ruby binding for WS-Management (rwsman) and some other sub-projects.

2.2 Scripting Language

One of the first decisions to be made is the selection of an appropriate scripting language. This decision must be made early in the design stage of the library, because of the "Object-Oriented Programming" goal. Binding of CIM objects must be natural for the selected language and there are some CIM/WBEM specifics that might be solved differently in different programming languages (e.g. some languages are unable to directly express abstract class – i.e. class which can be used only as a base class and not for creating instances). By selecting the scripting language first, it will be possible to create mapping that "looks and feels" natural to the developers.

CIM is object-oriented thus the scripting language must natively support Object Oriented Programming paradigm (classes, inheritance, polymorphism, etc). Being "popular"[11] in community of developers is another very important feature as the language must be easy to program in and it should provide a wide range of libraries and modules to make the development fast. Basically there are only three widely adopted, open source (see the "Openness" goal in Section 1.3) scripting languages satisfying these prerequisites. These are Perl[12], Python[13] and Ruby[14].

There are lots of (often very emotional) debates in different on-line discussion groups covering all advantages and disadvantages of different languages and still there is no simple answer. In Figure 2.7 you can see the result of Google Trends[15] for the query "perl, python, ruby". Although this result might be interpreted in many different ways it says that none of these languages is searched more often than the other. At least to some extent it means they are approximately equally popular.

There are also many essays discussing which programming language is the most suitable for a given project. Their conclusion is usually that "the only real

[11] Popular in this context means the language is often discussed in internet groups, there are many applications written in this language, a lot of supplementary libraries exists, major Linux distributions have an interpreter/compiler for this language installed by default, etc

[12] http://www.perl.org

[13] http://www.python.org

[14] http://www.ruby-lang.org

[15] "Google Trends analyzes a portion of Google web searches to compute how many searches have been done for the terms you enter relative to the total number of searches done on Google over time. We then show you a graph with the results – our search-volume graph – plotted on a linear scale." [26]; Google Trends is available on http://trends.google.com

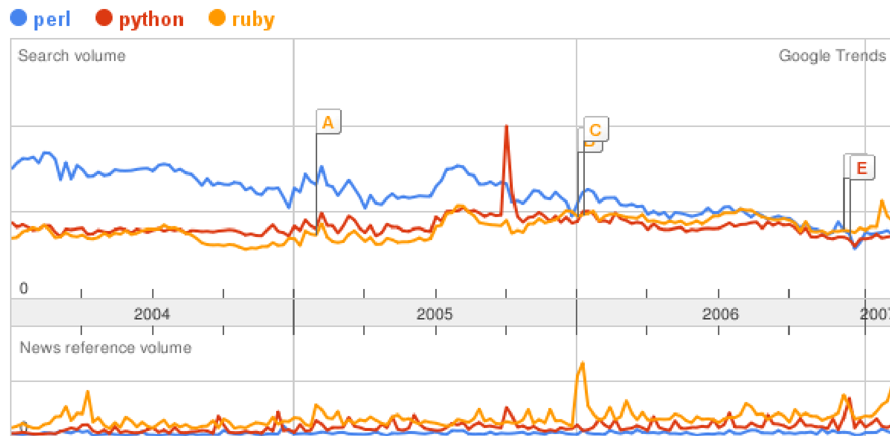


Figure 2.7: Google Trends - Perl, Python, Ruby [26]

difference is which one you know better" [24] or "there is no silver bullet in the world of programming languages" [25]. This is the main reason why Python is chosen as the scripting language for this work.

Another reason why to use Python (and not perl for example) is the fact Python interpreter comes with a very good interactive mode, which will make satisfying the "Shell Enablement" goal easier. IPython provides even better interactive support (more about this project can be found in Section 2.1.4).

2.3 Selected Python Features

Python is a high-level, interpreted language first released in the early 1990s by Guido van Rossum. It provides object-oriented paradigm together with high-level data structures (such as associative arrays) and dynamic features like introspection or reflection [31]. It has some interesting features and conventions that will make mapping CIM to Python easier and natural[16].

In Python 2.2 (at the time of writing this work the latest stable Python release is 2.5) so called new-style classes were introduced. The difference between old and new-style classes is beyond the scope of this work but it should be noted that not all features discussed here are available for old-style classes (which are still the default for compatibility reasons).

Another constructs were introduced to Python even later (e.g. decorators as defined in [33] were added to Python in version 2.4) and therefore Python version 2.5 is implicitly assumed in context of this work.

More about Python syntax and semantics can be found in [28] and [29].

[16]Please consult [29] for a Python introduction. This work expects that the reader is familiar at least with some basic Python constructs. It is beyond the scope of this work to provide an introduction to Python and at the same time [29] provides very nice tutorial.

The following features are of interest for this work:

Cross-platform. As long as a given platform has a Python implementation (i.e. the interpreter and the standard library) it can run Python code. Currently Python is available for all major operating systems – Windows, Linux/Unix, Mac OS etc. There are even versions that run on top of .Net or Java Virtual Machine environments.

Interactive mode. The most widespread implementation of Python (CPython) comes with an interpreter that supports interactive mode. This mode is often used by developers for quick experiments. It supports tab completion[17] what makes it ideal for fast explorations of libraries (see also "Docstrings"). Usually the primary prompt looks like the following example:

```
1  king:~ # Python
2  Python 2.4.2 (#1, Apr 13 2007, 15:45:45)
3  [GCC 4.1.0 (SUSE Linux)] on linux2
4  Type "help", "copyright", "credits" or "license" for more information.
5  >>>
```

Compound data structures. Python comes with a number of compound data structures. Most versatile types are list (dynamic array/sequence), tuple (an immutable sequence) and dictionary (also called associative arrays in other programming languages).

Docstrings. A convention heavily used by most (if not all) Python code ever written are so called docstrings. Each object (where object in this case might be any Python object – class, instance, method, etc) has a special string attribute called `__doc__` where documentation string is stored. This can be used either to generate documentation (HTML, PDF, etc) or to interactively read the documentation in interactive shell.

For example if a developer needs to find out how to use `append` method of a `List`, they can do the following in interactive mode:

```
1  >>> list = [1,2,3]
2  >>> list.append.__doc__
3  'L.append(object) -- append object to end'
```

Introspection. Introspection is a feature that allows a programmer to determine the type of an object at run-time. In Python it is used at different places – e.g. pydoc module is using introspection to generate html documentation

[17]Tab completion refers to a feature when a program automatically fills in partially typed commands [11].

```
def MyDecorator(fn):
    def new(*args):
        print "calling method"
        result = fn(*args)
        print "method returned"
        return result
    return new

@MyDecorator
def plus(a,b):
    return a+b
```

Figure 2.8: Decorator Example

of Python code from the docstrings. More about this feature can be found in [19].

Duck typing. Also called "latent typing" is a dynamic typing principle that determines the type of an object by introspection at run-time rather than by explicit relationship to some other object. As a consequence it is not important what is the type of an object, but what interface the object has.

This feature makes object-oriented programming in Python little bit different from programming in languages like C++ or Java. For example Python does not force the developer to upcast[18] their classes in case they need to use them as a parameter of a function and at the same time they are of different types. Bruce Eckel is discussing this feature more in depth in his essay "Strong Typing vs. Strong Testing" [22].

Decorators. Decorator is a function and a special syntax that will modify another function or method. In Figure 2.8 you can see a definition of a decorator called "`MyDecorator`" and how this decorator can be used to modify the "`plus`" function. In this case `MyDecorator` prints a message before and after calling the actual function. Therefore, calling the "`plus`" function will produce the following output:

```
>>> ten = plus(7,3)
calling method
method returned
>>> print ten
10
```

More about Python decorators can be found in [18].

[18]By upcasting it is understood going from a specific class/type to a more generic one.

Customized attribute access. It is possible to define how class attributes are accessed (i.e. set-ing, get-ing and delet-ing attributes). If a class has the following methods defined, they are used for accessing the attributes:

`__getattr__(self, name)` – In case the standard lookup did not find the attribute this method is called. It is supposed to return value of "`name`" attribute.

`__setattr__(self, name, value)` – set value of "`name`" attribute. "`value`" is the value being assigned.

`__delattr__(self, name)` – used for deleting an attribute.

`__getattribute__(self, name)` is called unconditionally (i.e. not after the standard lookup mechanism had failed as the `__getattr__` method) to implement attribute access.

Descriptors. A different way how to customize attribute access are so called descriptors. If a class property is an instance of a class with `__get__`, `__set__`, `__delete__` methods defined, these methods are used for computing/accessing the property (classes with such methods are also called descriptor classes).

Comparing with methods defined directly in the class (e.g. `__getattr__`), this has the advantage that property (or descriptor) classes might encapsulate some other logic specific just for a single property.

Class namespace implementation. Namespace of a Python class is implemented using a dictionary (associative array) accessible via `__dict__` attribute of the class. All attribute references are thus translated as a lookups to this dictionary – e.g. calling `C.x` is the same as calling `C.__dict__['x']` (where `C` is a class object)[19]. If the attribute is not found here the attribute search continues in the base classes.

This gives the developer more flexibility. For example it is possible to implement custom attribute access which will fall-back to the standard behavior under some conditions. The developer can also easily introspect the class for its methods and attributes by examining this dictionary.

Reflection and meta-classes. Reflection is a process that allows Python to modify the code being executed at run-time. For example it is possible to

[19]These two expressions are equivalent only if `x` is not a descriptor class instance and at the same time `x` is not an attribute of a parent class. If `x` is a descriptor class instance `C.x` will result in calling the `__get__`,method of the descriptor class (i.e. it is equivalent to `C.__dict.__['x'].__get__()`). `C.__dict__['x']` returns the descriptor class itself.

If `x` is defined in a parent class of `C`, it is stored in a `__dict__` attribute of the parent class.

add new classes or add/change methods and attributes of the class during the code execution.

Reflection is closely related to the concept of meta-classes. In Python, classes are objects too and as such they are also created by instantiating another objects – meta-classes.

In linguistic terminology, if classes are nouns and methods are verbs, meta-classes will be adjectives [2]. With meta-classes it is possible not only to create new classes at runtime (i.e. instantiate the meta-class) but also to define behavior of classes and their instances. Since classes are created from meta-classes, the respective meta-class controls how the class is created and it can even change the resulting class in a desired way.

The concept of meta-classes also allows the developers to use aspect oriented programming (AOP) to enhance classes with different capabilities.

More about meta-class programming can be found in [2], [16] and [17].

Chapter 3

Mapping CIM to Python

As discussed in Section 1.1, CIM consists of two parts – CIM Specification and CIM Schema. While CIM Specification defines the modeling language (meta schema), CIM Schema specifies a generic set of objects that are modeled in language defined in the CIM Specification. Expressing objects defined in the meta schema in Python (class, methods, ...) is the crucial part of this work.

As this work is focused solely on developing the client side of Common Information Model not all Named Elements (see Figure 1.1) must be mapped to Python. Specifically these Named Elements will not be discussed:

`Trigger` exists only on the server side (the client is working with `Indications`, which are objects created as a result of trigger).

`Schema` is used only for administration and class naming.

`Named Element` is the root of inheritance hierarchy. The developer works only with specialized `Named Elements` hence it is sufficient to discuss only the specialized elements[1].

In case a specific `Named Element` is defined by specialization (e.g. `Indication` is a specialized `Class`) everything discussed in the parent element applies also to the specialized elements unless otherwise stated.

3.1 Classes

Classes are the basic building and modeling blocks of CIM and thus their Python mapping is naturally the first step. Although it might seem that this mapping is straightforward there are still at least three ways how to do it:

[1]Named Element is the root of the inheritance hierarchy (i.e. each CIM object such as class or method is a named element) and thus it could be natural to map Named Element and its properties to Python and then enhance this mapping for the specialized elements. However, Named Element itself does not provide any features common for all objects and therefore this approach would not bring any benefits.

1. Create a generic class and store its properties (methods, qualifiers, ...) in well defined class attributes such as "`properties`" or "`qualifiers`". This is the option pywbem implements (see Section 2.1.2).

   With this approach the following construct would be possible:

   ```
   >>> print instance.properties['MyProperty']
   MyProperty Value
   ```

   In this case `instance` is an instance of some CIM class and `MyProperty` is a string property of this class. The value of `MyProperty` is "MyProperty Value".

2. Map each CIM class to a single Python class and customize attribute and method access (i.e. upon attribute access, internal structure will be searched and the correct value will be returned). In fact, this is just slightly enhanced to what pywbem does.

   Using this approach this construct would be possible:

   ```
   >>> print instance.MyProperty
   MyProperty Value
   ```

   In this case `MyProperty` is internally not a Python class attribute. The value of `MyProperty` is stored in the same internal structure as in the previous case (i.e. in the `properties` attribute). When the standard Python attribute look up fails, the customized attributed access returns the correct value.

3. For each CIM class, create a specific Python class with the same inheritance hierarchy, properties, methods, etc.

   In this case CIM properties will be mapped to Python class attributes and thus the following construct would be possible:

   ```
   >>> print instance.MyProperty
   MyProperty Value
   ```

   Note that in this case the standard attribute access provided by Python will be used.

Option number one is the simplest to implement but has most disadvantages. The developer needs to access special data structures with special names what makes them less productive and the development more demanding.

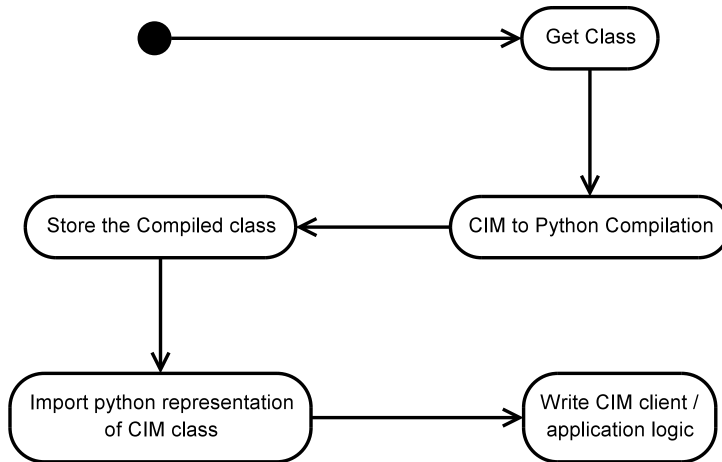


Figure 3.1: Get-Compile-Store-Load-Develop Cycle

It would be possible to implement the second alternative just by introducing methods for customized attribute access (see Section 2.3). When attribute is accessed a getter/setter method would check its internal data structure and either return or set the correct value. Comparing to the first option it will make programming easier but introspection of such objects would be still difficult because the developer would still need to know the format of the internal structure. As a result it would not be possible to use tab completion in interactive mode what makes the usage much more difficult.

Although the third option does not have the problems of former ones and at the same time it is the most intuitive, it is much more difficult to implement. It requires creating Python classes for every single CIM class (including the inheritance hierarchy) a programmer might want to use. There are several problems that make it difficult to implement.

First issue is that the CIM classes must be compiled into Python classes. A straightforward approach would be to compile them (from MOF format) to Python and store them in a module a CIM client developer can import and use. In context of this work this approach is referred as “static” compiling.

However, the client developer might not have access to the CIM classes beforehand. A possible CIM/WBEM usage is to get an instance object from a WBEM server at run-time and work with it. Such approach is very likely when accessing WBEM server from an interactive shell.

In Figure 3.1 you can see a possible activity diagram of development, in situation the developer does not have access to CIM class representation before implementing the client logic. It would require a few steps more for them – they would need to get, compile and store the classes in a module before they actually can write any application code.

Approach depicted in this figure would be required in statically typed languages such as C++[2]. However, Python is dynamically typed and "duck-typing" (see Section 2.3) is heavily used among Python programmers. It supports meta-classes thus it is possible to create classes at run-time (i.e. to instantiate a suitable meta-class).

Thanks to these dynamic features, it would be sufficient to compile CIM classes at run-time. With such approach it is possible for the programmer to skip the get/compile/store/load cycle completely and start developing the application logic immediately[3]. Compilation would be done at run-time and the developer does not need to know about it. This reduces the activity diagram from Figure 3.1 to a single "write application logic" state what makes development easier and faster. In context of this work this approach is called "dynamic" compiling.

Unfortunately compiling CIM classes at run-time is not possible when the WBEM server is unable to return a CIM class representation at run-time (e.g. the current proposal of WS-Management protocol does not yet define "get class" operation, hence this functionality is not implemented). In such case "static" compiling would be required. Although it would be possible to make some assumption about the class from its instance[4] this approach would be very limiting (e.g. it would not be possible to get the inheritance hierarchy, method names, qualifiers, etc). For these reasons both "static" and "dynamic" compiling of CIM classes is needed.

Although one-to-one mapping of CIM and Python classes is the most difficult to implement, it is possible and it is the only way how to satisfy goals defined in Section 1.3.

Associations

As you can see in Figure 1.1, `Association` is a specialized `Class`. Therefore, the same discussion as before applies for associations as well. Association classes have an extra qualifier (which is inherited by subclasses) indicating that this specific class is an Association. Unlike generic classes, associations may contain two or more references to other classes or class instances. Hence there are three use-cases how to access associations:

1. Enumerate instances of a given association class (e.g. `CIM_Dependency`) possibly with some filtering constraints.

[2] Unless some more sophisticated approaches are used.

[3] The developer will be heavily using "duck-typing" in this particular case. At the time of writing the application code, none of the CIM objects will exists and therefore it will not be (even theoretically) possible to perform any compilation time checks to make sure the developer is calling the correct methods or attributes.

[4] Instance in this case is a set of (PropertyName, PropertyValue) pairs

2. Given an instance, the user might want to get all or a subset of classes (instances) associated with this instance.

3. Given a class, the user might want to get all or a subset of classes (instances) associated with this class.

To cover these use cases a few methods will be introduced. CIM classes mapped to Python will have a static method called "`_GetClassAssociators`" and method called "`_GetAssociators`" to access class and instance associations respectively (note that the underscore at the beginning is there to distinguish these methods from methods modeled in CIM).

Enumerating instances of a given association class will be possible by using WBEM server operations – either by executing a custom query or by enumerating instances (see Section 4.1.3 for more details).

Indications

Like Association, Indication is also a specialized Class. Hence the rules that apply to generic class can be applied to Indication as well (i.e. Indication can have methods, properties, etc). There are no additional features of `Indications` that should be mapped to Python.

In order to support indications at the CIM client side, an HTTP listener (i.e. a program and/or function that will listen on a given TCP/IP port) must be implemented. This listener will receive WBEM messages with the indications and will dispatch them to the subscribed listeners. Listeners will implement a `IndicationOccured` method that will be called upon indication retrieval.

3.2 Properties

Although CIM properties can be directly mapped to Python class attributes (the terminology is different but properties and attributes represent the very same concept), there are some issues which should be solved.

As discussed in Section 2.3, each Python class has its namespace implemented as a dictionary object called "`__dict__`". This makes adding new attributes to class as easy as creating new element in the dictionary. However, CIM properties are complex objects with many different qualifiers. The simplest solution would be to create dictionary elements holding the value of the property. More sophisticated approach would be to create a descriptor class representing the property (more about descriptors in Section 2.3). Such class could easily hold other meta-data such as different qualifiers as well (more about property qualifiers in Section 3.4.3).

Descriptor class will provide a simple interface for the developer (i.e. accessing property values easily) and at the same time, the property will be represented

by a class storing additional information. Developers can still access directly the descriptor class using a special syntax (see Figure 3.4) when additional property attributes are needed.

3.3 Methods

Similarly to Properties, CIM methods can be directly mapped to Python methods with few modifications.

The first problem with CIM methods are their arguments because Python does not support calling by reference (i.e. function has access to argument values only)[5]. This makes implementation of output parameters as defined in CIM more difficult.

Using output parameters for returning more values from a function call is a common case for many CIM classes. However, Python is not the only language with this problem and similar works (such as CIM Client for Java or pywbem discussed in Section 2.1) must solve this problem as well. This is usually solved either by returning a special structure (as in the case of pywbem) or by passing a reference to a special "output" object which is altered inside the method call (this is implemented in CIM Client for Java).

Passing a reference to a special "output" object is not very convenient solution. The developer would be forced to instantiate this object before calling the method and to pass a reference of this object to the method call. The following construct would be therefore enforced:

```
1 >>> output = OutputVector()
2 >>> retValue = instance.MyMethod(1, 2, output)
```

The first line instantiates a special `OutputVector` structure. Line 2 calls `MyMethod` with two input arguments (`1`, `2`) and an output argument (`output`). The return value is stored in `retValue`.

On the contrary, method-call would be able to create and return a special structure automatically (it can be implemented in the mapping). The following construct will be possible:

```
1 >>> retValue,output = instance.MethodCall(1, 2)
```

Therefore this mapping will return output arguments in a special structure as a return value.

[5] Python is similar to Java in this case. All variables in Python are object references and thus a Python method has an access to the value of the reference.

In Python we have three possibilities, how to return output arguments from a function call in a special structure [20]. To demonstrate the concept and possible solutions a function which takes three arguments will be defined. Each argument should be increased by one and returned in an "output" argument. At the same time the function is supposed to always return 0 as an indication the calculation was executed without problems[6].

There are following ways how to return the output arguments:

Tuple. Returning a tuple (see Section 2.3) is the simplest way how to return more than one value. This method is simple and convenient way to return small and fixed number of values. In that case it is possible to use constructs like:

```
1  >>> def IncreaseByOne(a,b,c):
2  ...    return 0, a+1,b+1,c+1
3  ...
4  >>> a,b,c = (1,2,3)
5  >>> retValue,a,b,c = IncreaseByOne(a,b,c)
6  >>> print retValue,a,b,c
7  0 2 3 4
```

Unfortunately this will be little bit awkward in case the method needs to return a lot of values. The developer also needs to know the exact number of return values in order to "unpack" them correctly unless they want to introspect the tuple. It is not possible to give names to the return values and the developer needs to know the exact order if they want to interpret them correctly.

Dictionary. Another option is to return a dictionary with all output parameters. With this approach it is not needed to unpack a specific number of return values like in tuple. Instead, the function will return a constant number of return values:

```
1  >>> def IncreaseByOne(a,b,c):
2  ...   return 0,dict(a=a+1,b=b+1,c=c+1)
3  ...
4  >>> a,b,c = (1,2,3)
5  >>> retValue,values = IncreaseByOne(a,b,c)
6  >>> print retValue, values["a"], values["b"], values["c"]
7  0 2 3 4
```

[6] In this particular case it does not make much sense but in general, functions both return value and have output parameters.

This way, it is possible to return unlimited and variable number of named arguments easily.

Dictionaries are easy to create but at the caller side they are not the most convenient. For example if the developer wants to use an interactive shell, they would not be able to use tab-completion to auto-complete the names of the output parameters.

Custom Class Container. Using a custom class is very similar to using a dictionary to return the values (it has even more similarities internally, because the class namespace is implemented by a dictionary).

It requires a custom class definition, but it can be defined once and reused many times. Hence it is barely a disadvantage.

With a custom class container it is possible to use the following constructs:

```
1  >>> class ResultContainer:
2  ...   def __init__(self, **kwargs):
3  ...     self.__dict__.update(kwargs) # copy arguments to attributes
4  ...
5  >>> def IncreaseByOne(a,b,c):
6  ...   return ResultContainer(_ReturnValue=0, a=a+1,
7  ...                          b=b+1 ,c=c+1)
8  ...
9  >>> a,b,c = (1,2,3)
10 >>> retValue = IncreaseByOne(a,b,c)
11 >>> print retValue._ReturnValue, retValue.a,
12 ...        retValue.b, retValue.c
13 0 2 3 4
```

Although this example looks most complicated, it solves the issues of the previous ones. It is possible to return unlimited, variable number of named arguments and the developer can use this container interactively (i.e. tab completion in an interactive shell will work as well).

CIM models can be very complex and therefore it is not possible to make any assumptions how the methods will be modeled. Return output parameters in tuple is not convenient, because it is not possible to associate return values with their names. Hence the developer is forced to know the exact ordering. Dictionary solves this problem but it is difficult to introspect this structure from interactive shell. Custom class container solves also this issue and thus the output parameters will be mapped to a custom class container.

Additionally, Python allows the developer to name function arguments and therefore the mapping should support this as well.

The “IncreaseByOne” example would be modeled in CIM the following way:

```
1    [Description("Increase each argument by one.")]
2 uint32 IncreaseByOne( [IN(true), OUT(true)] a,
3                       [IN(true), OUT(true)] b,
4                       [IN(true), OUT(true)] c)
```

It will be natural to Python developers if the mapping will support the following constructs:

```
1 >>> a,b,c = (1,2,3)
2 >>> result = someObject.IncreaseByone(a,c=c,b=b)
3 >>> print result._ReturnValue, result.a, result.b, result.c
4 0 2 3 4
```

In case the parameter is output only (i.e. the `IN` qualifier is set to `false`) it should be omitted from the function call. It does not make sense to explicitly call a function with an argument that will be ignored. The actual output argument will be returned in the custom class container upon return from the function.

3.4 Qualifiers

In Common Information Model, qualifiers are used to characterize Named Elements (classes, methods, attributes, etc). Although some of the qualifiers can be easily expressed in Python most of them can be not. This section will discuss how qualifiers could be mapped and what to do with qualifiers that can not be directly expressed.

CIM Specification defines a set of standard qualifiers. However, this set can be extended by the user by defining custom qualifiers. Not knowing the semantics of these qualifiers beforehand, it is impossible to map them to Python. However, they still should be stored and programmatically accessed as any other qualifier.

Most of the qualifiers can be used to describe concrete Named Elements (e.g. it is possible to make aliases of properties, references and methods using the "`Alias`" qualifier). Qualifiers in this section are divided to groups according to Named Elements they are usually used with.

Nevertheless of the mapping, qualifiers must be stored so the CIM developer can access them at the programming (API) level. The "storage" method will also define how to access qualifiers that are not directly mapped to Python.

Looking at Figure 1.1, qualifiers must be stored for `Class`es, `Propert`ies and `Methods`. `Association`s and `Indication`s are specialized `Class`es thus it is possible to store their qualifiers the same way as for classes. The same applies for `Reference` which is a specialized `Property`. Although theoretically it is possible to use `Qualifier`s also to describe `Qualifier`s, it does not have any practical

use and therefore this option will be ignored[7]. `Triggers`, `Schemas` and `Named Elements` will not be discussed for reasons mentioned earlier (see introduction to Chapter 3).

Class Qualifiers are the easiest to store, although there are at least two approaches. The first and most straightforward solution would be to store qualifiers as a (Python) class attributes with a special naming convention (e.g. "`_qualifier_<NAME>`").

On the other hand, if there are a lot of different qualifiers (note that qualifiers might be inherited from base classes) this approach defines a lot of new properties in the class namespace. This might be slightly annoying while using tab-completion in an interactive shell.

This approach can be modified and a new class property `_QUALIFIERS` can be introduced, grouping all the class qualifiers in a single structure. In this case `_QUALIFIERS` will be a class, so the developer can use tab-completion to get values of qualifiers in the interactive shell.

A typical direct access to qualifiers (i.e. not via special mapping) in Python will be:

```
1 >>> print wbemManager._QUALIFIERS.Description
2 This class represents the OpenWBEM CIM object manager.
3 >>>
```

Property Qualifiers are not as straightforward as class qualifiers. At the same time it is desired to see properties as a simple, value-holding objects (to make the development easier) and to see them as more complex objects with their qualifiers and maybe some other characteristics (to make the scripts more sophisticated when needed).

As already discussed in Section 3.2, Python comes with the concept of descriptors making both "goals" achievable.

Descriptor is a custom class representing a single property. As such, it can provide any functionality a generic Python class provides. However, accessing the descriptor class (and not the value it represents) requires a special syntax (see Figure 3.2, line 5), which is not very convenient. For this purpose it is needed to define a convenience method `_GetPropertyQualifier` in the class the property belongs to. In that case construct as shown in Figure 3.2 (line 3) will be possible.

The first print statement (line 1) prints the value of the "`Name`" property. The second print statement (line 3) prints the "`Description`" qualifier of

[7] Even the MOF language syntax does not have any way how to express qualifier's qualifiers.

```
1 >>> print wbemManager.Name
2 OpenWBEM:74513e6b-0860-11dc-8000-fb6e6048eec7
3 >>> print wbemManager._GetPropertyQualifier('Name','Description')
4 The Name property is used to uniquely identify a CIM Server.
5 >>> print wbemManager.__class__.__dict__['Name'].Description
6 The Name property is used to uniquely identify a CIM Server.
```

Figure 3.2: Accessing Property Qualifiers

"`Name`" property. The last print (line 5) shows the direct access to descriptor class properties (call at line 3 is a shorthand for call at line 5)[8].

Method Qualifiers can be stored similarly to class qualifiers, directly in the method object as attributes (methods in Python are "`instancemethod`" objects)[9]. As in classes, there are exactly the same two options how to store these qualifiers. Each qualifier can be stored as a property of the method object with a special naming convention or a single property will exists, grouping all qualifiers for the given method.

To make the development easier it will be best to store the qualifiers consistently[10] and therefore the same concept of a special `_QUALIFIER` attribute will be used for methods as well. Following the convention this construct will be possible:

```
1 >>> print wbemManager.StopService._QUALIFIERS.Description
2 The StopService method places the Service in the stopped state.
3 >>> wbemManager.StopService()
4 0
5 >>>
```

The first statement (line 1) prints the description of `StopService` method. The second statement (line 3) executes the method and successfully stops the service (return value is 0 – line 4).

All class, property and method qualifiers will be encapsulated in special class as discussed above. This will allow the developers to use the same syntax as in the examples and at the same time it will be possible to access some additional meta-data about the qualifier (such as its type).

[8]Note that if `Name` attribute would be defined in the parent class of `wbemManager` instance, this syntax would not work.

[9]The only difference is that attributes of method objects are read-only. However, qualifiers can not be modified anyway thus this is not a limitation for the mapping.

[10]That is, to store them the same way as the classes and properties store their qualifiers – in a special class.

Nevertheless of the Python to CIM mapping, all qualifiers defined in Common Information Model will be stored as described in this section. Additionally, some qualifiers might be stored at other places (e.g. a copy of `Description` will be also available in the `__doc__` property) or provide additional constraints (e.g. `Abstract` qualifier might prevent the developer from instantiating the class).

3.4.1 Generic Qualifiers

Qualifiers discussed in this section can be used to describe any Named Element defined in the CIM meta schema (Figure 1.1).

Description. This qualifier provides description of the named element.

Classes and methods in Python have a special convention of a "`__doc__`" attribute which can be used for storing the documentation strings (see Section 2.3 for more details about this feature). Hence the mapping is straightforward in this particular case.

Situation with the Property element is more complicated. Although "docstrings" for properties were discussed in the past by the Python community (see [32]), there is no way how to express them at the moment.

Some tools (e.g. epydoc) provide their own convention to document attributes. However, this convention can be used only when parsing the source-code file and it is not possible to access such documentation in interactive shell. However, the advantage is that the generated documentation is nicely structured and the developer can find the information they need fast and very easily. There are overview sections for "Instance Methods", "Class Variables" and sections with detailed method and property documentation. Unfortunately these conventions are not standardized by the Python community and therefore it is not possible to use standard tools to generate the API documentation.

Another solution is to append the "Description" of a property to the corresponding class docstring. This would be the most "portable" solution. It will work both in the interactive shell (the developer could simply use the class docstring to get the documentation of all properties available) and with different tools generating API documentation (such as epydoc or pydoc). Unfortunately the disadvantage is that it makes the documentation less structured (e.g. it is not be possible to get the documentation of a single, concrete attribute – just one huge string with all the documentation for the class and all its attributes).

The last option how to map Description qualifier to docstring, is to use descriptor classes (see Section 2.3 for more information). As discussed in Section 3.2, properties will be represented by descriptor classes anyway.

```
>>> print service.Started.__doc__
bool(x) -> bool

Returns True when the argument x is true, False otherwise.
The built-in True and False are the only two instances of the class bool.
The class bool is a subclass of the class int, and cannot be sub-classed.
```

Figure 3.3: Incorrect Access to Descriptor Class Docstring

Since descriptors are regular Python classes it is possible to attach the property docstring directly to the corresponding descriptor instance. Unfortunately storing docstrings directly in descriptor class has some issues as well.

The first problem is that this documentation is not processed correctly by all API documentation tools[11] although this can be considered as a bug of these tools.

Second problem is that it is not easy to access descriptor docstrings and special syntax is needed. The code snippet in Figure 3.3 does not print the documentation of `Started` property of a `CIM_Service` class as one might expect. It prints documentation of a `bool` data type instead. The correct way how to access the documentation is shown in Figure 3.4.

The first and natural approach does not work, because of the way how descriptors work. They are hiding the fact a property is a complex object and they return only the value of such property. Therefore in Figure 3.3 a boolean value is returned and only later the `__doc__` attribute is accessed.

On the contrary, when the developer knows an attribute is hidden in a descriptor class, they can use the approach shown in Figure 3.4.

To make accessing these docstrings easier, it is possible to provide convenience method such as `_GetPropertyDocumentation(propertyName)` in the class namespace. In IPython it is possible to introduce custom, so-called "magic", functions, to make the access even more convenient from interactive shell environment (for more information see Section 4.6).

Despite the drawbacks (mainly the portability among API documentation tools), implementing the property docstrings via `__doc__` attribute of a descriptor seems to be the best alternative.

Deprecated, Experimental, Expensive are qualifiers which are notifying the

[11] Pydoc which comes with standard Python library can process such documentation without any problems. Epydoc does not process descriptor class docstrings correctly.

```
>>> print service.__dict__['Started'].__doc__
Type: boolean

Started is a Boolean that indicates whether the Service has been
started (TRUE), or stopped (FALSE).
```

Figure 3.4: Accessing Docstring of Descriptor Class

developer something strange might happen when accessing such objects[12]. As they are only notifying it does not make much sense to create any special constraints for them. However, it will be useful to see the access to such elements in the log files during the debugging phase (e.g. entries such as "calling expensive method "ExpensiveMethod"" or "getting an instance of deprecated class "CIM_Deprecated"").

The standard ([4]) defines also other qualifiers that can be used for any named element. These are `Displayname`, `Mappingstrings`, `Modelcorrespondence` and `Provider`. These will not be mapped to any Python construct, but they can be accessed by the developer as described in Section 3.4. For more information about these qualifiers see [4].

3.4.2 Qualifiers Specific for Classes

With some exceptions, this section summarizes qualifiers which are specific only for `Classes`. If a qualifier can be applied to another `Named Elements` (e.g. methods), mapping to the specific element is discussed as well.

Abstract, Exception, Terminal. These three qualifiers[13] define constraints on inheritance and instantiation of classes. There is no way, how to express them in Python natively, because Python does not support these constructs.

Although it would be possible to simulate the correct behavior by custom meta-classes (see also Section 2.3) there are few reasons why this is not needed or even desired in some cases.

These constraints are checked by the CIM providers at the server side anyway. In case a developer tries e.g. to instantiate an abstract class, the

[12] Although the names of these qualifiers should be self-explanatory, it is possible to find their exact definition in [4].

[13] Abstract class is a common construct in object oriented languages. In CIM, Exception class is defined the same way as abstract class with some additional constraints (e.g. the Exception qualifier is inherited). Terminal class is a class that can have no subclasses. More information can be found in [4].

corresponding provider should throw an exception[14] which will be delivered to the developer. Hence the developer will be notified no matter what the mapping is. Implementation of these qualifiers will be therefore redundant. Extra code also means extra bugs.

In one particular case this implementation even might not be desired. This Python mapping could be used to easily write testsuites (unit tests) for the CIM providers. In such case it is desired to try forbidden constructs at the script level to check the correct behavior and exception handling of the CIM provider.

As a result these qualifiers will be just appended to the documentation string of the corresponding class. The developer can find these constraints in documentation and no extra checks at runtime will be performed.

OCL. Object Constraint Language (OCL) is a formal language used to express constraints on UML models [10]. These constraints are usually invariants that must be true for the modeled entity. In CIM, OCL expressions can be used for classes and methods.

Although it would be possible to implement the OCL support for Python objects, it is far beyond the scope of this work. In some cases it would not even be possible to check the constraints at the Python level and only the CIM provider level is able to perform them. At the same time, arguments used for not implementing the Abstract, Exception and Terminal qualifiers can be used. Therefore OCL expressions can be accessed via standard means but no additional Python mapping and checking will be done.

This qualifier will be mentioned in the docstring of the corresponding class or method, though.

Invisible. Classes, properties and methods with the invisible qualifier are defined only for internal purposes. As such they should not be displayed to the user nor relied upon.

Invisible qualifier will be mentioned in the docstring of the corresponding object (class, property or method).

Large. This qualifier indicates that a class or a property requires a large amount of storage space.

[14] Please note that there are two different types of exceptions discussed at this place. CIM classes with `Exception` qualifier are custom exceptions specific for the model. At the same time, provider (or WBEM server) can throw run-time exceptions defined in [5]. For example in this particular case (i.e. trying to instantiate an abstract class) a "CIM_ERR_FAILED: Abstract methods can not be instantiated by the provider" exception might be raised. WBEM server exceptions are discussed in Section 4.1.5.

The developer should be aware of this fact and therefore it should be mentioned in the docstring of the object. On the other hand it is not desired to create additional runtime constraints (such as generating a confirmation message before actually accessing the affected class or property).

However, during the development the developer might want to log accesses to such objects. A message will be written to the logs (such as "Accessing large property "LargeProperty" of the "CIM_MyClass" instance").

Class qualifiers `Revision`, `Version` and `TriggerType` can be accessed as described in Section 3.4. They will not be mapped specifically to Python.

Association Specific Qualifiers

The following qualifiers are specific for association classes:

Delete, Ifdeleted. These qualifiers indicate what must be done with objects in the given association in case some of them are deleted. Since this is implemented on server side it does not need to be mapped to Python.

Aggregation, Composition. These qualifiers indicates that an association is aggregation or composition as defined in UML Specification from OMG. It does not need to be mapped to Python, because the object associations are always fetched from WBEM Server and thus they should work as implemented at the server side.

3.4.3 Property, Parameter and Method Qualifiers

Many qualifiers can be used for properties, method parameters and methods (their return values) at the same time. Therefore, they are discussed together in this section. In case the qualifier can not be used for a specific Named Element, it is clearly stated in the text. Please also note that some of the attributes were discussed in previous sections.

Constraints Qualifiers

All qualifiers discussed here are limiting what can be done and what values are valid for a given property, parameter or method. Hence they will be mapped similarly.

One way how to map these qualifiers would be to use extra Python code performing the constraints checking at runtime. However, as discussed in Section 3.4.2, these checks would be just redundant to what the CIM provider is doing. The developer will be notified by an exception anyway thus the only difference is when and by which component will it be generated.

This leads to conclusion that adding these qualifiers to the docstring of a corresponding object to notify the developer will be enough.

Specifically these qualifiers will be mentioned in the API documentation of the CIM classes:

Maxlen, Minlen. These qualifiers specify the maximum and minimum length, in characters, of a string element.

Maxvalue, Minvalue. Maximum and minimum value of an (integer) element.

Read, Write. Indicates that a property can be read or written. This qualifier can be used only for properties.

Required. Indicates that a non-NULL (i.e. valid) value is required. `Required` can be used only for properties and references.

Qualifiers Specifying the Data Type

There are several qualifiers that specify in more detail what values can the developer expect in the given element. At least some of these should be mapped to Python.

Arraytype. For an array, `Arraytype` specifies the characteristics of the array (e.g. "bag" means that the same index to the array may return different values, while "indexed" means the same index always returns the same value no matter what operations are performed).

In Python, a list will be used to map CIM arrays. This type corresponds to "indexed" `Arraytype`. Indexed arraytype is the most restrictive and thus it trivially includes all other arraytypes making them useless in Python.

However, the developer should be aware of the fact, that CIM provider implementation may behave differently than what they can see in the Python binding. This is not a problem because the developer should be aware of the arraytype, when using arrays (and there is no way how to hide this information to make the development easier – for more information see [4]).

Arraytype can not be used for methods.

Valuemap, Values. These two qualifiers are a concept similar to enumerations (the `enum` keyword) or static constants in C++ and Java.

`Valuemap` defines a set of valid values while `Values` provides translation between integer values and strings. Although these two qualifiers can be used independently, they will be usually used together to both define the set and the corresponding strings.

In case the developer wants to define a set of constants for a class property, method parameter or a return value, it is a common practice to define static

constants in the corresponding class. Therefore static constants are the best fit for mapping `Valuemap` and `Values` qualifiers.

Properties, methods and their parameters are always defined in a class and therefore it is clear where the static constants should be introduced – in the namespace of the corresponding class. Creating a static constant for each valuemap element would result in a class with a lot of properties (there can be hundreds of constants defined for a class and these constants are inherited). This would make interactive shell (and specifically tab-completion) very difficult to use.

These reasons lead to a solution that will group all constants in a single attribute visible in the class namespace – `_CONSTANTS` (similarly as `_QUALIFIERS` discussed in Section 3.4). This attribute will group all constants for all properties and methods defined in the class (the same applies for method parameters, but the constants will be defined in the method object namespace – this concept was already used to store qualifiers as discussed in Section 3.4).

As a convention the following naming scheme will be used for the constants (all names will be capitalized to highlight the fact they are constants): `<NAME_OF_THE_ELEMENT>_<VALUE_NAME>`. `NAME_OF_THE_ELEMENT` is the name of a property or a method for which `Values` qualifier exists. `VALUE_NAME` is the string name of the property defined in `Values`. In case the `Values` element does not exists but the `Valuemap` is defined (i.e. a set of valid values is defined, but no string mapping exists) the numerical value will be used instead (e.g. `MYPROPERTY_10`). Note that the opposite situation (when `Values` is defined but `Valuemap` is not) is not a problem – the CIM Specification defines that an index of the `Values` array (zero-indexed) should be implicitly used as the integer value otherwise being defined in `Valuemap`.

Using this schema the following constructs will be possible:

```
1 >>> if wbemManager.OperationalStatus == \
2 ...     wbemManager._CONSTANTS.OPERATIONALSTATUS_STARTING:
3 ...          print "Service is starting..."
```

In this example, the `OperationalStatus` property of a `CIM_Service` class is used to check the status of the `CIM_Service` instance. This property defines different statuses using the `Valuemap` and `Values` qualifiers.

Nullvalue. Nullvalue defines a value that indicates the item to be NULL (i.e. not having meaningful value). This is similar to defining custom constants using the `Values` qualifier. Hence this qualifier can be mapped to a class property using the following schema in the `_CONSTANTS` namespace: `NULL_<PROPERTY_NAME>`.

This construct can be used to check for NULL:

```
>>> if wbemManager.OperationalStatus == \
...     wbemManager._CONSTANTS.NULL_OPERATIONALSTATUS
...         print "Operational status is not defined"
```

Nullvalue qualifier can be used only for properties.

Bitmap, Bitvalues. Bitmap indicates which bits are significant in a bitmap. Bitvalues provides translation between bit position and associated string (similar concept as `Values` qualifier).

It is difficult to create a mapping for this qualifier without knowing a typical use-case. The developer may want to use this qualifier to perform some bitwise operation (such as AND or OR) – in such case they will probably need to have this mapping in the format of pre-calculated numbers (i.e. if bit 10 is significant they may want to have "1024" pre-calculated).

At the same time they may want to know the bit positions (i.e. the format provided by CIM).

Therefore, these qualifiers will be mapped to a special class property called `_BITVALUES` the same way as `_CONSTANTS` or `_QUALIFIERS` are. They will store the number of significant bit using the following naming schema: `<PROPERTY_NAME>_<BITVALUE>`. `PROPERTY_NAME` indicates the name of a property and `BITVALUE` is the string representation from `Bitvalues` qualifier.

According to standard these two qualifiers must be always used together[15].

DN. DN qualifier can be used only with string types and it specifies that the value must be a distinguished name as defined in a given standard[16]. This qualifier will not be mapped to Python.

Octetstring. Octetstring signalizes that a given property, method or parameter is an octet string (i.e. data whose length is a multiple of eight). Similarly to the `DN` qualifier, this qualifier will not be mapped to any Python construct.

Embeddedobject, Embeddedinstance. These two qualifiers were added to the standard quite lately as a workaround for not being able to express objects (classes, instances) in properties, parameters or methods. In the next major version of CIM specification it is expected that a new data type for embedded objects will be introduced [4].

[15] The standard says: "The number of entries in the BitValues and BitMap arrays MUST match." [4]. This leads to a conclusion that if one of these qualifiers is non-empty, the other must be non-empty as well.

[16] For more details please consult [4], page 16 – the `DN` qualifier definition.

Currently it is possible to encode objects and instances to strings. Unfortunately, it is not precisely defined which encoding should be used and the standard allows both MOF and CIM-XML encodings to be used depending on the usage. It means, that the developer must be aware they are working with an embedded object. Hence the Python mapping can not be too sophisticated and it should not translate string to and from Python objects automatically. Otherwise it could choose the wrong encoding and the code may not work.

Therefore these objects will be stored as they are – in a string. Python objects will provide convenience methods to easily express them in both of these formats (`_toMof`, `_toCimXML`).

Unknown values, Unsupported values. These qualifiers are specific for properties and in concept similar to `nullvalue` qualifier. There are two semantic differences, though.

First, these qualifiers define set of values instead of one single value as in the `nullvalue` case.

Second, they define values which are unknown and unsupported respectively. In both cases it means that the property can not be considered to have meaningful value.

These sets will be mapped the same way `nullvalue` is, in the `_CONSTANTS` namespace of the corresponding class. The following convention will be followed: `UNKNOWNVALUES_<PROPERTY_NAME>` and `UNSUPPORTEDVALUES_<PROPERTY_NAME>` for unknown and unsupported values respectively. This property will hold a list of values. Hence the following construct will be possible in Python:

```
1 >>> if wbemManager.OperationalStatus in
2 ...     wbemManager._CONSTANTS.UNKNOWNVALUES_OPERATIONALSTATUS
3 ...         print "Operational status is not known"
```

Semantic Qualifiers

Some of the qualifiers define how a given property, method or a parameter are (or can be) used:

Gauge. Represents an integer that may increase or decrease in any order of magnitude.

Counter. Represents a non-negative integer that monotonically increases.

Units. Defines the units of the associated data item. A complete list of standard units can be found in [4], Appendix C.

Semantics of this qualifiers is not enforced by any means by the CIM/WBEM infrastructure (i.e. it is up to the CIM provider implementation to do some additional checks). On the other hand, the developer should be aware of modified semantics and thus this should mentioned in the API documentation (i.e. Python docstrings).

Qualifiers Specific for Methods

There are no qualifiers specific only for methods. Qualifiers that can be used with methods are discussed in this Section, in Section 3.4.1 (Generic Qualifiers) and in Section 3.4.2 (Qualifiers Specific for Classes).

Qualifiers Specific for Parameters

The only two qualifiers specific for parameters are `In` and `Out`. `In` specifies that the parameter is used as an input value. `Out` parameters are used for returning values back to the caller. Since these two qualifiers are very closely related to calling the methods, they are discussed in Section 3.3 in more depth.

Other Qualifiers

Key. Qualifier specific for properties and references only. It specifies that a property is needed to uniquely identify instance of the given class (think of primary key in relational databases).

It is important to explicitly state this in the programmers documentation, so the developer can easily find all key properties. No other mapping is needed.

Propagated. Property specific qualifier. It defines a name of a propagated key (if a property is propagated it must be a key). This qualifier is important when designing the class and similarly to key, no specific mapping is needed. More information about this qualifier can be found in [4].

Static. Static methods and properties can be accessed without having an instance object (i.e. class object is enough). Mapping of this qualifier is straightforward.

In Python, all properties can be thought as static (this is a consequence of Python's data/object model). For static methods the `staticmethod` decorator can be used.

Alias. Alias provides another name under which an item (property, method or reference) can be accessed. In Python, the item will be duplicated so the developer can use both names the same way.

Schema. This qualifier only states the name of the schema. It can be accessed the same way all other qualifiers and no further mapping is needed.

Propertyusage. Propertyusage is another, property specific qualifier. It (optionally) allows to classify, how properties should be interpreted and used. It is sufficient to store the value of this qualifier in the property descriptor.

Override. This qualifier specifies that a given property, method or reference overrides a similar construct in the ancestor class. In python this is done automatically when the class defines the same property/method, with the same signature as its ancestor class. Hence no specific mapping is needed.

Syntax, Syntaxtype. According to [4], syntaxtype defines a format of the syntax qualifier. Syntax defines a specific type assigned to a data item. Unfortunately the standard is very brief and it is not clear how are these qualifiers used and how to map them to Python. No classes defined in CIM Schema are using them at the moment of writing this work.

Qualifiers Specific for References

The following qualifiers are specific for references: `Max`, `Min`, `Delete`, `Ifdeleted` `Weak`, `Aggregate`.

Only the `weak` qualifier will be mentioned in the docstring of the corresponding reference (property). The other qualifiers are either informational or provide some additional constraints and thus they will not be mapped to Python. Please consult [4] for more information.

3.5 Intrinsic Data Types

CIM Schema ([4]) defines also a basic set of intrinsic data types that can be used for class properties, method parameters, qualifiers and so on.

Python provides the following data types: integer (32 bits), long (arbitrarily large integer), float (64 bit, double precision), boolean, complex and string (ASCII/EBCDIC or unicode). Character in Python is represented by a string of one character.

Data types defined in [4] are similar to what Python provides. CIM is also able to express integers, floats, boolean, strings, characters or datetime data types. Although Python does not have directly datetime type, the standard library ("`datetime`" module) provides objects that can be used for this purpose. Other data types can be directly mapped to a corresponding Python type. However, the precision in Python might be different from the precision in CIM (e.g. Python does not support 8 bit integers). In such cases always the higher precision is used (e.g. 8 bit integers from CIM will be mapped to 32 bit integers in Python).

Chapter 4

Implementation

Chapter 3 provides all information needed to implement the mapping of Common Information Model to CIM. However, it still must be decided, how the code will be structured, which technologies should be used (e.g. which XML libraries to use to implement CIM-XML protocol) and some other technical details. The prototype implementation is called powerCIM.

As already discussed in Section 3.1, mapping from CIM to Python can be performed either during the execution of the CIM client code (referred as "dynamic" compiling) or before the client code is executed (i.e. compiling from MOF to Python). Both approaches have their advantages and there are cases when one or the other will be preferred. However, the compilation itself is just a technical problem once the mapping (as discussed in Chapter 3) is defined. For the purpose of this work the "dynamic" compiling (i.e. at runtime from xmlCIM to Python) is implemented as there are some interesting concepts that can be used (for example reflection and meta-programming).

On the other hand, MOF language can be expressed as LL(1) parseable grammar (see [4], Appendix A) and therefore it should not be difficult to implement the "static" compiler as well.

4.1 High-level Overview

In Figure 4.1, you can see an UML diagram of the powerCIM Python package. Classes depicted on this diagram and their usage will be discussed in this section.

These classes are split into several Python modules and `__init__.py`[1] file is provided for importing the whole powerCIM package easily.

The code is structured in the following way:

`__init__.py` – initialization of the package. All powerCIM modules are imported and the logging subsystem is initialized.

[1] `__init__.py` is required when a package is stored in a directory. Python is looking for this file to initialize the whole package (usually there are just imports of all important modules).

factory.py – implementation of a `WBEMFactory` and `ClassFactory` (internal class implementing the mapping from CIM to Python) classes.

cim.py – implementation of different mapping constructs. This modules implements `CIMClass`, different descriptor classes (`Property`, `Qualifier`) etc.

meta.py – implementation of meta classes.

wbemconnection.py – implementation of the `WBEMConnection` class and its subclasses (i.e. the communication with WBEM server via TCP/IP network).

testing/ is a directory which contains unit tests developed for powerCIM. More about these tests in Section 4.5.

examples/ is a directory with some sample scripts developed using powerCIM. Some of these examples are discussed in Appendix A.

IPython/ contains configuration files for IPython integration. More about them in Section 4.6.

4.1.1 WBEMConnection and its Subclasses

Besides the CIM to Python mapping, the communication with WBEM server is one of the first problems that must be solved. Web-Based Enterprise Management defines more communication standards. Currently these are CIM-XML and WS-Management (although this standard is in a preliminary status for about a year). However, the developer should not be bothered with connection details and all they need to know is the protocol they want to use.

For communicating with WBEM server WBEMConnection class is defined. This class serves as a base class for different communication protocols and it defines a set of basic operations such as `GetClass`, `GetInstance` or `ExecuteQuery`. Each method defined in this class raises `NotImplementedError` exception. Each subclass must therefore implement this functionality, otherwise the developer using powerCIM will get an exception.

For the purpose of powerCIM only the CIM-XML protocol is implemented in `CIMXMLConnection` class using the pywbem project (see also Section 2.1.2)[2]. This makes implementation of the communication part very easy. `CIMXMLConnection` class will delegate the methods to pywbem, returning the internal structures to `WBEMFactory` (discussed later). `WBEMFactory` takes care of the mapping as discussed in Chapter 3.

[2]As discussed in Section 2.1.2, pywbem provides both the CIM-XML implementation and some very basic mapping of CIM constructs to Python structures. The format of these structures is also used in powerCIM. PowerCIM is using pywbem as its back-end both for operations and its internal structures.

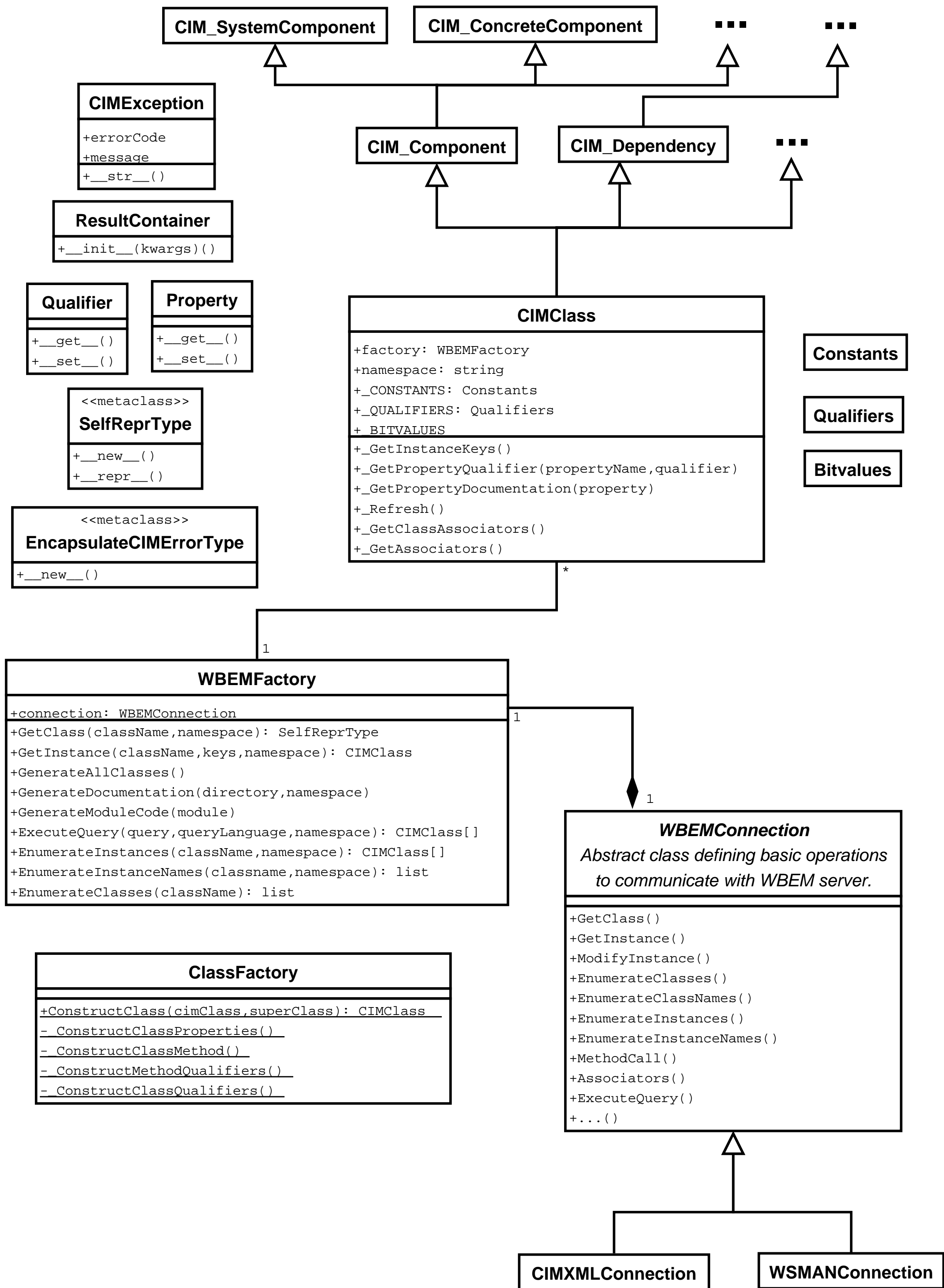


Figure 4.1: PowerCIM Overview

4.1.2 CIMClass and ResultContainer

CIM Specification does not require CIM classes to have an inheritance predecessor. However, having a common interface for all CIM Classes mapped to Python will make it possible to introduce several convenience methods common for all mapped classes (such as method for accessing property qualifiers as discussed in Section 3.4). In powerCIM, `CIMClass` is the root of inheritance hierarchy of all CIM classes mapped to Python.

`CIMClass` also defines how values of class properties will be fetched from WBEM server. There are two ways, how to access them:

1. At each access, get the value of a property directly from WBEM Server.

   This has the advantage, that the values are up to date at the time they are accessed (on the other hand there is still a possibility that value was changed while transferring it to the developer or while the developer actually uses it for some computations).

2. Get the whole instance at once at the time the Python class instance is created and provide interface for refreshing the whole instance at once.

   This approach is less network intensive (each WBEM message has a lot overhead as the communication is usually in XML) comparing to the previous and thus the user experience may be better. On the other hand it may happen that the developer needs just a single property and they will still need to get the whole instance via network.

Both of these options are reasonable and depending on the usage they may be useful. However, for the purpose of this prototype implementation, the second option is implemented (i.e. getting all properties at once). It would be easy to improve this implementation and provide a special flag, that will switch the behavior (i.e. getting properties each at once).

Similar question arises when a value is assigned to a class property[3]. In this case property can be either assigned and the instance directly modified at WBEM server side, or a special "commit" method call will modify all instance changes at once.

Unlike read, assignment may have side effects. For example changing an `IsEnabled` property of a CIM class to `True` might start a service and change another properties of this instance. Therefore this prototype implements a solution which modifies the instance after each property assignment. An automatic refresh of the instance at the client side is performed upon successful modification.

This implementation has the advantage that the developer can use instances in read-only mode also during the time the network connection is broken.

[3]Properties in CIM can be marked as writable using the `write` qualifier discussed in Section 3.4.3.

In powerCIM, `CIMClass` is the single interface common for all CIM classes mapped to Python. The developer is always able to call the following methods (note that names of these methods are prepended by underscore character ("_") to differentiate them from methods defined in the modeled CIM class):

_Refresh() will get an up to date version of the current instance. That is, it will re-read all property values.

_GetPropertyQualifier(propertyName, qualifier) returns a value of a given `qualifier` for the given `propretyName`. This is a convenience method as discussed in Section 3.4.

_GetPropertyDocumentation(propertyName) is a convenience method for getting the docstring of a property.

Note that although the `Description` qualifier is mapped to the docstring (i.e. the `__doc__` attribute), it is not the same as the docstring. The docstrings are comprised from other qualifiers as well (e.g. there may be a note that a given property is `key`).

_GetClassAssociators(factory, namespace, assocClass, resultClass, role, resultRole) is a static method that returns a list of classes associated to this class (not to this instance). For this purpose a reference to a `WBEMFactory` instance (the "`factory`" argument) and optionally a namespace ("`namespace`") is needed.

The remaining arguments are optional and they serve as filters for the set of classes being returned. Semantics of these arguments is as follows:

assocClass must be a valid CIM association class name. Each returned class (instance) will be associated to the source class (instance) via this class or one of its sub-classes.

resultClass must be a valid CIM class name. Each returned class (instance) will be this class or one of its sub-classes (instance or instance of one of its subclasses).

role must be a valid property name. Each returned class (instance) will be associated to the source class (instance) via an association in which the source plays the specified role (i.e. the property name must match).

resultRole must be a valid property name. Each returned class (instance) will be associated to the source class (instance) via an association in which the returned class (instance) plays the specified role.

_GetAssociators(namespace, assocClass, resultClass, role, resultRole) returns a list of instances associated with this instance.

All arguments of this method are optional. `namespace` is a namespace of the associated instances. Other arguments can be used for filtering the resulting set. They are discussed in the `_GetClassAssociation` method.

_GetInstanceKeys() returns a dictionary with `(key, value)` pairs of the current instance. This dictionary uniquely identifies this instances in a given namespace and WBEM Server.

_GetModelPath() returns a model path of this instance as defined in [4]. E.g. it may return the following string:
"Xen_VirtualSystemManagementService.CreationClassName=Xen_VirtualSystemManagementService,SystemName=king.suse.cz"

_GetObjectName() returns object name of this instance as defined in [4]. E.g. it may return the following string:
"king.suse.cz/root/cimv2:Xen_VirtualSystemManagementService.CreationClassName=Xen_VirtualSystem-ManagementService,SystemName=king.suse.cz"

Each `CIMClass` instance has a reference to the `WBEMFactory` (`factory` attribute) by which it was created. Therefore all instances can live on their own and they do not need any special parameters when used (such as URL or login and password when calling a method). Although factory itself is associated with a namespace, each `CIMClass` instance must know their own (implemented in the `namespace` attribute), as `WBEMFactory` methods can be called with arbitrary namespace.

CIM instance is stored in internal structure in the "`instance`" attribute.

`Constants`, `Qualifiers` and `Bitvalues` are another three important classes related to `CIMClass`. They are used to store values and nullvalues (as discussed in Section 3.4.3), qualifiers (Section 3.4) and bitmaps (Section 3.4.3) respectively. However, they are used only as inner classes and get never instantiated. In this case class is used as a structure to store static constants. Inheritance hierarchy of these classes copy the same structure as the class in which they are included (i.e. if a `CIM_Component` is a superclass of `CIM_SystemComponent`, the `_CONSTANTS` class of `CIM_SystemComponent` is a subclass of `CIM_Component`'s `_CONSTANTS` class).

As discussed in Section 3.3, output parameters of methods will be mapped to a container class. This container is provided by the `ResultContainer` class. It is a simple class with custom constructor that can take a dictionary as an argument. Elements of this dictionary will be translated to class attributes at the time of instantiating the container (i.e. for each key in the dictionary an attribute with the same name will be created holding the dictionary value).

4.1.3 WBEMFactory and ClassFactory

`WBEMFactory` is the most important class for the CIM client developer. It is an interface which enables them to communicate with a WBEM Server. It provides

methods for getting instances, enumerating instances, executing queries etc.

Although internally powerCIM uses internal structures to store CIM objects, `WBEMFactory` is creating objects as defined in Chapter 3 (Mapping CIM to Python). For mapping CIM to Python, `WBEMFactory` uses an internal factory class – `ClassFactory`. This class implements the mapping as defined in Chapter 3.

The developer will instantiate the factory with a WBEM Server URL and credentials (login and password) and optionally with a namespace (if namespace is not specified, `root/cimv2` will be used) and a communication protocol (currently only CIM-XML is supported). Once the `WBEMFactory` is instantiated they can use the methods to access WBEM Server functionality.

Although most of the methods provided by `WBEMFactory` can be called with an explicitly stated namespace, by default, the namespace specified at instantiation of the factory is used. This makes the factory very convenient to use, because once an instance is available, the developer does not need to care about URLs, credentials or namespaces.

The connection to a WBEM Server is stateless (HTTP is used as a backend protocol both for CIM-XML and WS-Management). As a consequence, `WBEMFactory` instance will be able to operate even if the WBEM Server was restarted meanwhile.

`WBEMFactory` provides the following methods:

__init__(url, login, password, namespace, module, protocol) is constructor of WBEMFactory.

`url` is the URL[4] of WBEM server the developer wants to connect to (e.g. https://localhost:5989).

`login` and `password` are used to authenticate the client at WBEM server.

`namespace` is the default namespace which should be used for all operations (such as EnumerateInstances). If not specified, `root/cimv2` is used.

`module` is a name of module which contains "pre-compiled" mappings of CIM classes (see also the `GenerateModuleCode` method). This module must be already imported.

`protocol` is the protocol that should be used to connect to WBEM server. At the moment this argument is ignored and CIM-XML is used because no other protocols are implemented.

GetClass(classname, namespace) returns a Python mapping of the given CIM class (`classname`). If the class is accessed for the first time, it is fetched from WBEM server and it is mapped to a Python class. Once the

[4]Uniform Resource Locator

mapping is finished, the Python class is stored in an internal caching dictionary. Next time the same class is being accessed, the mapped class is fetched from this internal structure.

Optionally a namespace can be specified. If namespace is not specified, the value given at factory instantiation is used.

GetInstance(classname, keys, namespace) returns an instance of the specified `classname` identified by the `keys` (`keys` is a dictionary where the dictionary key is name of the key and the dictionary value is the respective value of the key). Optionally a namespace can be defined. This method returns always an instance of `CIMClass` sub-class. The developer can directly call methods or access properties of such an instance as specified in Chapter 3.

GenerateAllClasses(namespace) does not return anything. This method fetches all available classes from WBEM server, maps them to Python classes and stores them in an internal cache. If a class is being accessed next time, the mapping does not need to be performed and thus the `GetClass` method is faster.

GenerateDocumentation(directory, namespace) generates a pydoc documentation in the given `directory` (the default value is "./html/") for all available classes in the given namespace (`GenerateAllClasses` method is called within this call).

GenerateModuleCode(moduleName, namespace) generates a python module of the given name (`moduleName`). The generated module will contain the mappings for all available classes within the given namespace. The module can be imported so the mapping is not performed always at runtime (making the code little bit faster). It can be also used for generating documentation by different API documentation tools (i.e. not only pydoc, which is used by `GenerateDocumentation` method, but also tools like epydoc).

Please note, that this code is generated and it is supposed to be neither readable nor comprehensive.

ExecuteQuery(query, queryLanguage, namespace) will execute a `query` expressed in the given query language (`queryLanguage`) and `namespace`. Return value is a list of instances of `CIMClass` sub-classes (i.e. a list of CIM classes mapped to Python).

Unfortunately it is not very easy to determine what query languages are supported by the WBEM server. Although [5] defines ways how to determine the server capabilities[5], even one of the major open-source server

[5]See [5], Section 4.5: Determining CIM Server Capabilities

implementation, openWBEM, is unable to give the developer a list of supported query languages.

Therefore, selecting the correct[6] query language is responsibility of the developer.

EnumerateInstances(className, namespace) returns a list of instances of the given `className`. Although it would be possible to implement this method by executing a query of the form "`SELECT * FROM className`", this is not desirable as executing queries may be not supported by the WBEM server (while intrinsic method EnumerateInstances usually is supported). However, semantically these two approaches are identical.

EnumerateInstanceNames(className, namespace) returns a list of dictionaries. Each dictionary consists of three keys – `classname`, `keys` and `namespace`. Since these keys and their format (`classname` is string, `keys` is a dictionary with keys and `namespace` is a string) is the same as the `GetInstance` method expects, they can be directly used for getting the instance using the following Python construct:

```
1 >>> serviceName = factory.EnumerateInstanceNames('CIM_Service')[0]
2 >>> service = factory.GetInstance(**serviceName)
3 >>> print service.Name
4 OpenWBEM:74513e6b-0860-11dc-8000-fb6e6048eec7
```

EnumerateClasses(className, namespace, ...) returns a class of the given `classname` and all their subclasses in a list (i.e. the return value is a list of `CIMClass` descendants). Each class in this list is the Python representation (mapping) of the corresponding CIM class. The mapped classes are stored in the same cache as classes created by `GetClass` method and the same rules apply for them – if the mapping is already in cache, it is not computed again.

This method accepts some other parameters (the corresponding WBEM method accepts them – see [5], Section 2.3.2.9 – EnumerateClasses) but in most cases the developer does not to care about them and the default values should work fine.

GetConnection(), GetNameSpace(), GetURL() are three getter methods for accessing the connection class (`WBEMConnection` instance), namespace and URL of this factory respectively.

[6]Correct in this case means supported by the WBEM server.

4.1.4 Qualifier and Property

`Qualifier` and `Property` are descriptor classes as discussed in Section 2.3.

`Qualifier` is an descriptor class for storing and encapsulating qualifiers (see Section 3.4). Internally, this class contains a pywbem structure for storing the qualifier. Qualifiers are read only and thus it is not possible to change them and the descriptor class does not implement the `__set__` method (in matter of fact, this method is implemented and it raises a "`QualifierWriteError`" exception when the value is assigned).

`Property` is a descriptor class as discussed in Section 3.2. Similarly to `Qualifier`, it also stores the property and all its meta-data in an internal structure. However, properties may be also writable (`write` qualifier) and thus there must be a mechanism for setting the property.

Descriptor classes are always bound to a `CIMClass` instance and therefore they have access to the WBEM connection. Upon write they can easily call the `ModifyInstance` method to change the property of the managed element at the server side.

After the instance is modified an implicit `_Refresh` is called, because assigning a property might have side-effects and the instance may change. Thus the instance is up to date after the assignment.

There is a question what to do with read-only properties (by default properties are read-only). It will be very easy to provide an additional check before calling the `ModifyInstance` method and raise an exception when the property can not be written. On the other hand, the exception will be risen anyway by the WBEM server little bit later. As discussed in Section 3.4.2 (the discussion about Abstract, Exception and Terminal qualifiers) checking this at the powerCIM level might prevent developers to create testsuites for CIM providers. Therefore the check will not be implemented (see also Section 5.1).

4.1.5 SelfReprType and EncapsulateCIMErrorType

`SelfReprType` and `EncapsulateCIMErrorType` are the only two meta-classes defined in powerCIM[7].

`SelfReprType` is used as a meta-class for `CIMClass` and all descendant classes. It is also used to create new CIM classes at runtime (i.e. the new CIM classes are instances of `SelfReprType` meta-class).

`SelfReprType` adds an important characteristic for each of these classes – they can represent themselves using the `__repr__` method. This method returns a string with a Python code[8]. Execution of this Python code will create a class

[7]For more about meta-classes see Section 2.3, [2], [16] and [17].

[8]However, the generated code still relies on the fact that powerCIM library exists and is imported.

with the same properties and methods. This feature is heavily used when generating Python module with mapped classes (`GenereateModuleCode` method of `WBEMFactory` class).

Another interesting methods defined in this meta-class are `_tomof()` and `_toxml()` used to express CIM classes either in mof or xml format (i.e. they return a MOF or XML string). These methods are defined both in `SelfReprType` and in `CIMClass`. This makes it possible to have the same methods (the same names) defined both in class (as "static" methods[9]) and instance (as a "regular" methods).

`EncapsulateCIMErrorType` is a meta-class used only for `CIMXMLConnection` class. It is used to implement a construct found in aspect oriented programming – each `CIMXMLConnection` method is encapsulated in a try/except block catching the CIM exceptions generated by pywbem. Unfortunately pywbem exceptions are not very explanatory (although they return error code and additional description it is usually not enough and the developer needs to know what the error code represent). Therefore they are mapped to custom exception class – CIMException. This class provides more explanatory exceptions and developers should catch these exceptions in the client code.

4.2 Extrinsic Methods

As discussed in Section 3.3, CIM Class methods will be mapped directly to Python methods in the namespace of the corresponding class. However, there are still some technical difficulties that must be solved.

When the developer wants to call a method on a `CIMClass` instance, they want to call it like any other method on any other instance (i.e. with some parameters and they expect a return value). As already discussed in Section 3.3, the method signature in Python must be different from what can be found in the MOF declaration of the same class and method. The reason for this are output parameters which will be returned together with return value in a tuple.

Extrinsic method is constructed using an `ExtrinsicMethodCall` method defined in `ClassFactory`. This method will return a function that can be directly stored in the corresponding class namespace. The method itself will check the input parameters (whether all input parameters are specified) and in case of an error a standard `TypeError` exception will be raised (i.e. the standard Python behavior is simulated).

If the input parameters are correct, the extrinsic method call will be performed and the instance will be refreshed using the `_Refresh` method (by calling a method the instance might change and thus an up to update instance is needed).

[9]Although in this case, these methods can be called without having a class instance, they are not exactly the same as static methods. A class instance of the `SelfReprType` meta-class is needed.

Once the method call finished it is time to encapsulate the output parameters in the `ResultContainer` class and return them with the return value in a tuple. In case the method has no output parameters, only the return value is returned (so the developer does not need to care about unpacking the tuple).

The method will also contain a special `__repr__` method that will be used by the method to represent itself in a Python code.

All other qualifiers, constants and similar constructs as discussed in Sections 3.3 and 3.4.3 will be added to the constructed method by the `ClassFactory`.

4.3 Logging and Debugging

Python standard library includes a module called `logging` which provides all logging functionality powerCIM needs. Therefore it is used as logging subsystem for powerCIM.

This module is extremely configurable and the developer is able to choose from a wide range of logging handlers (such as FileHandler for logging to files or SMTPHandler for sending the log messages to a chosen e-mail address). It also provides a few predefined log levels (such as DEBUG, INFO or CRITICAL). For more information about this module, please consult [30], Section 14.5 – "logging – Logging facility for Python".

Since most of the operations are performed using network it would be useful for the developers to log all network accesses for debugging purposes. For these reasons all `WBEMFactory` methods are logging DEBUG level messages when accessing a remote functionality (e.g. getting an instance, enumerating instance names, etc).

Additionally operations on an `CIMClass` are logged as well – setting a property, calling a method and getting associated classes and/or instances are operations that are logged.

As defined in Chapter 3, some qualifiers (such as `Expensive`) will generate log messages as well.

The logging subsystem is initialized in the `__init__.py` file. The configuration is stored in "`~/.powerCIM/logging.conf`", where "`~`" represents the home directory of the user currently using the powerCIM library. If this file does not exists it will be created. The default configuration is shown in Appendix B.

By default, `WBEMFactory` (`logger_factory`) and `CIMClass` (`logger_CIMClass`) are writing their `INFO` log level messages to a file (`/var/log/powerCIM.log`). At the same time these messages are written to console (`logger_root`) as well.

4.4 API Documentation

As discussed in Chapter 3, a lot of information, including some of the qualifiers, are stored in Python docstrings for classes, methods and properties.

This makes the Python mapping a great source of documentation and the developers have this documentation in a form they are used to.

There are several ways, the developer can access the documentation:

Directly from interactive shell. It is possible to access directly the `__doc__` property of the object to access the docstrings or use a special syntax provided by IPython (putting a question mark after a class or method name will print the documentation string; it is also possible to use a "magic" method for accessing property docstrings – this will be discussed in Section 4.6).

Using an API documentation generation tool. It is possible to generate the documentation both at runtime (using pydoc from the standard python library) or store the classes in a module file. In that case is is possible to use any tool available for generating Python code documentation (e.g. epydoc).

4.5 Unit Tests

During the development a few unit tests were developed (they can be found in the "`testing`" directory). These tests are based on `unittest` module provided by Python standard library. They are trying to exercise most of the mapping functionality provided by powerCIM.

During the development they proved to be a useful way, how to find bugs in powerCIM code especially after refactoring.

All tests share the same configuration (WBEM server URL, credentials and namespace) and they are divided into the following groups:

Factory tests are testing the basic functionality of `WBEMFactory`. Specifically creation of the (mapped) classes, enumerating instances and classes, etc.

Class tests are performing class specific tests. In this case getting classes associated with a given class (i.e. stressing the `_GetClassAssociators` function).

Instance tests are performing tests on real CIM instances to check the mapping.

Query tests perform some execute query tests.

Linux_OpertingSystem tests are testing `Linux_OperatingSystem` class functionality.

4.6 Integration With IPython

```
1  conn:~$ ipython
2  powerCIM.factory:INFO:  WBEMFactory created, url=http://localhost, login=, ↩
3  password=*****, namespace=root/cimv2
4  Python 2.5 (r25:51908, May 25 2007, 16:14:04)
5  Type "copyright", "credits" or "license" for more information.
6
7  IPython 0.7.2 -- An enhanced Interactive Python.
8  ?       -> Introduction to IPython's features.
9  %magic  -> Information about IPython's 'magic' % functions.
10 help    -> Python's own help system.
11 object? -> Details about 'object'. ?object also works, ?? prints more.
12
13 In [1]: s = factory.ExecuteQuery('SELECT * FROM CIM_Service WHERE Name="sshd↩
14 "')[0]
15
16 In [2]: s.Started
17 Out[2]: True
18
19 In [3]: docu s.Started
20 Type: boolean
21
22 Started is a Boolean that indicates whether the Service has been
23 started (TRUE), or stopped (FALSE).
24
25 In [4]:
```

Figure 4.2: IPython as CIM Enabled Management Console

As already mentioned in Section 2.3, IPython is an interactive, object-oriented shell based on Python. Although the standard Python implementation comes with an interactive shell, IPython is more powerful and provides many features the standard interactive mode does not have. It can be used both to develop and debug applications and as a system management console.

If somebody wants to use the interactive shell that comes with Python by default, they can use powerCIM as any other Python module (by executing the "`import powerCIM`" or "`from powerCIM import *`" command).

IPython provides some extra options how to integrate powerCIM functionality better.

If powerCIM should be used as a CIM enabled system management console, it would be nice to have a configuration which will import powerCIM and set up other properties of IPython upon start.

This can be easily achieved by a custom ipythonrc file in the ~/.ipython/ directory. Example shown in Appendix C, Section C.1 shows an ipythonrc file that imports powerCIM and creates a WBEMFactory instance connected to localhost with no login and no password. It also executes powercim-magic.py (see below).

Using this configuration the developer or system administrator can directly access WBEM server as shown in Figure 4.2. In this example a CQL query is executed (line 13) to get a sshd service instance. Line 16 shows that the sshd service is currently started. With no extra typing, the system administrator was able to find out the current status of ssh daemon.

In IPython it is possible to create so called "magic" commands to extend the basic set. These functions allows the user to control IPython itself and also to call many system related commands (such as ls to get a list of files in the current directory in unix systems).

In Section C.2, you can see a definition of magic function called "docu". This function calls _GetPropertyDocumentation method for the given object (it works both for methods and class attributes). Usage of this magic function is shown in Figure 4.2 on line 19.

Chapter 5

Summary

This work defined a mapping of Common Information Model to Python and successfully implemented a prototype in Python called powerCIM. The prototype implementation is capable of fetching classes from WBEM Server and map them dynamically (i.e. at run-time) to Python objects. At the same time, many convenient methods are provided (e.g. to work with associations or for accessing qualifiers of different objects) making the development in Python and CIM very easy.

Choosing Python as a scripting language proved to be an important decision for several reasons. Python's philosophy is "batteries included" [34] and therefore it comes with a powerful standard library covering a wide area of tasks (such as XML parsing, e-mail handling, multimedia services, etc). Python is also a modern language implementing many interesting dynamic features (as discussed in Section 2.3) making the mapping of Common Information Model to Python possible or at least easier and more convenient.

Synergy with Python standard library and other open-source projects is also very important. Additional and standard libraries provide an extremely powerful framework for the developers. It is possible to develop graphical applications using the Python Qt[1] binding, create web application using mod_python Apache[2] module or create command line applications.

CIM enablement of Python provided by powerCIM puts these synergies into even broader context. With powerCIM it is possible to create management applications with different user interfaces very rapidly. They can be anything from command line driven, web based (thin client) up to sophisticated graphical interface (thick client).

Comparing with other projects (see also Section 2.1), powerCIM delivers an easy to use, intuitive API for developing CIM clients. Although powerCIM is

[1]Qt is a cross platform GUI toolkit for application development written in C++. Python Qt binding (PyQT) can be downloaded from http://www.riverbankcomputing.co.uk/pyqt/.

[2]An open-source HTTP server implementation. mod_python can be downloaded from http://www.modpython.org/.

based on pywbem, it enhances it and solves all problems discussed in Section 2.1.

Considering CIM Client for Java, powerCIM delivers a solution that can be used for experimenting with new ideas without the usual edit-compile cycle. With powerCIM it is also much easier to access class properties and call methods because CIM classes are mapped directly to Python classes with their attributes and methods[3].

Although SBLIM wbemcli could be considered as a project suitable for CIM scripting it is much more difficult to use than powerCIM as demonstrated in Figure 2.4.

VBScript and Windows PowerShell provide the same features as powerCIM. Both of these projects are mapping CIM classes to native classes and they can be used similarly as the mapping provided by this work.

On the other hand, powerCIM comes with some features that are not available in VBScript or Windows PowerShell. For example, it is possible to create CIM API documentation using epydoc, or to trace the network calls (such as getting class, enumerating instances, etc) using the logging subsystem.

Major advantage of powerCIM when comparing to tools from Microsoft, is the fact that it is an open-source project (powerCIM is released under LGPL[4] license). If needed, it is possible to enhance it very easily. Improving tools from Microsoft is impossible due their proprietary nature.

The goals defined in Section 1.3 were addressed as follows:

Object-Oriented Programming. This goal was one of the most important and the whole Chapter 3 deals with it. PowerCIM provides an object-oriented framework that can be used to develop CIM clients in Python using the OOP paradigm. At the same time Python conventions were followed whenever it was possible.

Documentation. The API documentation of powerCIM is provided by this work (Chapter 4) and, at the same time, it is possible to use Python doc-strings to get the documentation in an interactive shell.

Docstrings are provided both for classes defined by powerCIM and classes mapped to Python from CIM. It is also possible to use tools like pydoc or epydoc to generate documentation which can be stored in different formats (such as html).

Debugging. As discussed in Section 4.3, all network operations can be logged. PowerCIM provides a very sophisticated way how to configure the logging subsystem and the developers are able to configure how much information they want (different log levels) and how they want to store these logs (in a file, sending them via e-mail, etc).

[3]Calling a method in Java requires code shown on lines 24 and 25 in Figure 2.6. Calling a method in powerCIM requires much simpler code shown on line 50 in Section A.1.

[4]http://www.gnu.org/licenses/lgpl.html

Use of Scripting Language. Python was selected as a scripting language keeping (not only) this goal in mind. For more information see Section 2.2.

"Openness". PowerCIM is released under LGPL license. Although it was not tested extensively on different platforms it should be portable without bigger problems. The open-source natures makes it possible to fix any platform-dependent issues.

Anyone with Internet access can download the latest source code from `http://en.opensuse.org/PowerCIM`. The source code is also available on a DVD which is part of this work.

"Shell Enablement". The standard Python interactive shell can be used as a basic system management console. The mapping defined in Chapter 3 kept this goal in mind and features like "tab-completion" or Python docstrings were enabled. It is possible to get more sophisticated management console by using IPython as discussed in Section 4.6.

Goals of this work as defined in Section 1.3 were addressed and powerCIM should be an easy to use, object-oriented extension of Python with debugging and documentation support. With other open-source tools it is also possible to use it as a system management console.

5.1 Open Issues and Future Work

Each work can be extended and this one is not an exception. There are few open issues that were either not addressed or they are beyond the scope of this work.

First challenge are the tools for generating API documentation. Both mainstream tools – pydoc and epydoc have some issues. While pydoc can create documentation even when the developer is using more sophisticated constructs (such as descriptor class), the generated documentation is not structured nicely and not very easy to read.

On the other hand, epydoc creates very nice documentation but it does not handle all constructs correctly (e.g. descriptor class docstrings as discussed in Section 3.4.1).

Another open issue are embedded objects and instances. Unfortunately, this was added to CIM only lately and the usage is not very clear. It is expected that next major revision of the standard will define a new data type for embedded objects making this concept easier to understand and implement [4]. This is something that needs to be addressed in the future.

Besides these topics there are few other improvements that can be made:

WS-Management Support. The only supported protocol at the moment is CIM-XML (XML over HTTP). Having support for WS-Management ser-

vices would improve powerCIM significantly. As web services and specifically SOAP are very common these days it should not be difficult to implement WS-Management support.

Better Code Generation. Although powerCIM is able to save the CIM classes mapped to Python into a reusable module this functionality could be improved. Currently, the generated module is one huge file with all available class definitions. One approach could be to divide the classes according to their namespace and save one namespace per file.

MOF to Python Compiler. PowerCIM implements “dynamic” (for more information see Section 3.1) compilation of CIM classes at runtime. However, this approach can not be used in every circumstance. In some cases, it is needed to get Python definition of CIM class before connecting the CIM client to the WBEM Server. In such cases, a MOF to Python compiler would be needed.

Constraint Checking. As discussed in Section 3.4.3, there are few qualifiers that are putting some constraints on the objects they describe (e.g. maxlen, minlen, read, write, etc). There are several reasons why this checking should not be performed at powerCIM level in general. However, some developers might appreciate such feature under specific circumstances. In that case it would be nice to have this checking implemented with a special flag turning it on or off for a given Python module.

Creating Classes and Instances. CIM/WBEM defines a way how to create new classes and instances in a given namespace. Being able to sub-class `CIMClass` (see Section 4.1.2) and give this class to powerCIM might be useful in some cases (although it is probably not a typical use-case).

Similarly it might be useful to create instances from the CIM classes mapped to Python and let powerCIM to create such instances at WBEM server side.

Indications Support. PowerCIM is currently unable to receive Indications. To enable Indications it would be needed to implement a simple “server” that will listen to indications at a specified (TCP/IP) port.

Service Location Protocol Support. One of the standard defined by DMTF is “WBEM Discovery Using the Service Location Protocol” [8]. Support for SLP discovery would be especially useful when using powerCIM as a system management console.

Appendix A

PowerCIM Examples

Source code of examples discussed in this Appendix can be found on the DVD which is part of this work (see also Appendix D) in the `MFF/powerCIM/examples/` directory[1].

A.1 Start/Stop Services – Thick Client

This thick client example[2] demonstrates how CIM clients with graphical user interface could be developed using powerCIM and PyQt projects. This simple application can be used to easily start or stop services on a given operating system.

This example also demonstrates that powerCIM is easy to use because only very limited code is related to Common Information model and most of the programming deals with the user interface.

Lines 9-12 in the code listed below define the URL and credentials of WBEM server being used.

[1] If you are using the live system, the source code can be found in `~/linux/powerCIM/examples`. For more information see Appendix D.

[2] The source code can be found in `guiServices.py` file.

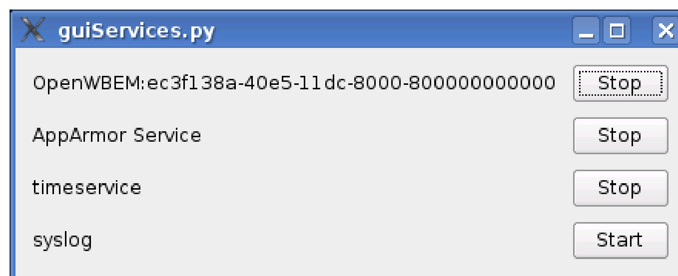


Figure A.1: guiServices.py Screenshot

Code on lines 71-73 connects to WBEM server and enumerates all instances of the `CIM_Service` class. For each instance of this class, one line in a grid (`QGridLayout` – line 77) with its name (line 85) and a start/stop button (line 86) is created. `StartStopButton` (line 14) is a custom button which is aware about the service it represents. Each `StartStopButton` contains a reference to a `CIM_Service` instance. If the button is clicked, the service is either started or stopped (depending on its previous state) and the button label is changed accordingly.

You can see a screenshot of this application in Figure A.1.

The rest of the code is Qt specific and is not directly related to powerCIM or Common Information Model.

```
1  #!/usr/bin/env python
2
3  import sys
4  from qt import *
5  from qttable import *
6  import powerCIM
7
8
9  host = 'http://localhost'
10 login = ''
11 password = ''
12 namespace = 'root/cimv2'
13
14 class StartStopButton(QPushButton):
15     """Custom push button aware of the fact that it represents a CIM_Service
16     status.
17
18     Each StartStopButton has a reference to a corresponding CIM_Service
19     instance.
20
21     It defines a public slot which can be used to change the status of the
22     service.
23     """
24
25     def __init__(self, service, parent):
26         """__init__(self, service, parent) -> Initialize the StartStop button.
27
28         Arguments:
29             service -- CIM_Service instance (its powerCIM mapping)
30             parent -- parent widget of this button.
31         """
32
```

```
33          if service.Started:
34              apply(QPushButton.__init__,
35                  (self,"Stop",parent))
36          else:
37              apply(QPushButton.__init__,
38                  (self,"Start",parent))
39          self.service = service
40
41
42      def slotChangeStatus(self):
43          """slotChangeStatus(self) -> change the status of the current service.
44
45          If service is started, it stops the service.
46          If service is stopped, it starts the service.
47          """
48          try:
49              if self.service.Started:
50                  self.service.StopService()
51                  if not self.service.Started:
52                      self.setText("Start")
53              else:
54                  self.service.StartService()
55                  if self.service.Started:
56                      self.setText("Stop")
57          except powerCIM.CIMException,e:
58              QMessageBox.critical(self,"guiServices.py",e.__str__())
59
60
61  class MainWindow(QMainWindow):
62      """ The main widget of the application.
63      It is a grid with two columns -- first column is the name of the service
64      and the second column is a push button which can be used to start/stop the
65      service.
66      """
67      def __init__(self, *args):
68          #call the super class constructor
69          apply(QMainWindow.__init__, (self,)+args)
70
71          #connect to WBEM Server and enumerate all CIM_Service instances
72          factory = powerCIM.WBEMFactory(host, login, password, namespace)
73          services = factory.EnumerateInstances('CIM_Service')
74
75          #create the grid
76          self.main = QWidget(self)
77          self.layout = QGridLayout(self.main,services.__len__(),2,10)
```

```
78          i = 0
79
80          #for each service in services create a new label with its name and a
81          #new StartStopButton. Add this button to the grid and connect the
82          #clicked() signal and changeStatus slot. This will change the status
83          #of the service each time the button is clicked.
84          for service in services:
85              self.layout.addWidget(QLabel(service.Name,self.main),i,0)
86              button = StartStopButton(service,self.main)
87              self.layout.addWidget(button,i,1)
88
89              self.connect( button, SIGNAL("clicked()"),
90                            button.slotChangeStatus)
91              i+=1
92
93          self.setCentralWidget(self.main)
94
95
96
97  def main(args):
98      #Qt related logic
99      app = QApplication(args)
100     win = MainWindow()
101     win.show()
102     app.connect(app, SIGNAL("lastWindowClosed()"), app, SLOT("quit()"))
103     app.exec_loop()
104
105
106 if __name__=="__main__":
107     main(sys.argv)
```

A.2 AppArmor Security Event Notification

This example deals with Novell AppArmor technology[3].

AppArmor is capable of sending e-mails when security access violations have been detected. Three types of notifications exists (Terse, Summary and Verbose) and each one can be configured separately. More about AppArmor notifications can be found in [36] in Section 4.2 – "Setting Up Event Notification".

AppArmor comes with CIM providers enabling system administrator to set-up these notifications using CIM and WBEM protocols. Example listed in this section defines two methods – `printNotificationSettings` (line 12) and

[3]Novell AppArmor is a Linux application security framework included in SUSE Linux products. More about AppArmor can be found in [35] and [36].

```
linux@king:.../powerCIM/examples - Shell - Konsole
Session  Edit  View  Bookmarks  Settings  Help
linux@king:~/powerCIM/examples> ./apparmor_report.py
powerCIM.factory:INFO:  WBEMFactory created, url=http://localhost, login=, passw
ord=*****, namespace=root/cimv2
===   AppArmor Security Event Notification   ===
        Notification is enabled.

        Terse Notification:
                Frequency: 600
                Email Address: user@somehost.com
                Severity: 0
                Include Unknown Severity Events: True

        Summary Notification:
                Frequency: 600
                Email Address: user@somehost.com
                Severity: 0
                Include Unknown Severity Events: True

        Verbose Notification:
                Frequency: 1800
                Email Address: user@somehost.com
                Severity: 0
                Include Unknown Severity Events: True
===   AppArmor Security Event Notification   ===
        Notification is enabled.

        Terse Notification:
                Frequency: 600
                Email Address: root@localhost
                Severity: 0
                Include Unknown Severity Events: True

        Summary Notification:
                Frequency: 600
                Email Address: root@localhost
                Severity: 0
                Include Unknown Severity Events: True

        Verbose Notification:
                Frequency: 1800
                Email Address: root@localhost
                Severity: 0
                Include Unknown Severity Events: True
linux@king:~/powerCIM/examples>
Shell  Shell No. 2
```

Figure A.2: An Example of apparmor_report.py Output

`changeNotificationsEmail` (line 46).

Both methods take an instance of `Novell_AppArmorNotificationSettingData` class as their first argument. Given this argument, `printNotificationSettings` prints the current configuration of notification settings (you can see an example of the output in Figure A.2).

`changeNotificationsEmail` takes an additional argument – a string containing e-mail address. This method configures AppArmor to send all notifications (i.e. Terse, Summary and Verbose) to the given e-mail address.

The code snippet listed below calls `printNotificationSettings` to print the current notification configuration. Then it changes the e-mail to `root@localhost` and finally prints the configuration again to demonstrate the e-mail addresses were really changed. Output of this script is shown in Figure A.2.

```
1  #!/usr/bin/env python
2  import powerCIM
```

```
3
4  # define WBEMServer
5  WBEMServer = 'http://localhost'
6  login = ''
7  password = ''
8
9  #get factory
10 factory = powerCIM.WBEMFactory(WBEMServer, login, password)
11
12 def printNotificationSettings(notify):
13     """printNotificationSettings(notify) -> print current settings of Apparmor
14     notification settings.
15
16     Arguments:
17        notify - Novell_AppArmorNotificationSettingData instance
18     """
19
20     print "===   AppArmor Security Event Notification   ==="
21     enabled="disabled."
22     if notify.IsEnabled:
23         enabled="enabled."
24
25     print "\tNotification is "+enabled
26
27     print "\n\tTerse Notification:"
28     print "\t\tFrequency: "+notify.TerseFrequency.__str__()
29     print "\t\tEmail Address: "+notify.TerseEmail.__str__()
30     print "\t\tSeverity: "+notify.TerseLevel.__str__()
31     print "\t\tInclude Unknown Severity Events: "+notify.TerseUnknown.__str__()
32
33     print "\n\tSummary Notification:"
34     print "\t\tFrequency: "+notify.SummaryFrequency.__str__()
35     print "\t\tEmail Address: "+notify.SummaryEmail.__str__()
36     print "\t\tSeverity: "+notify.SummaryLevel.__str__()
37     print "\t\tInclude Unknown Severity Events: "+notify.SummaryUnknown.__str__()
38
39     print "\n\tVerbose Notification:"
40     print "\t\tFrequency: "+notify.VerboseFrequency.__str__()
41     print "\t\tEmail Address: "+notify.VerboseEmail.__str__()
42     print "\t\tSeverity: "+notify.VerboseLevel.__str__()
43     print "\t\tInclude Unknown Severity Events: "+notify.VerboseUnknown.__str__()
44
45
46 def changeNotificationsEmail(notify, email):
47     """ changeNotificationsEmail(notify, email) -> Change all notification
```

```
48      emails to a given address.
49
50      Arguments:
51          notify - Novell_AppArmorNotificationSettingData instance
52          email - email address
53      """
54      notify.TerseEmail=email
55      notify.SummaryEmail=email
56      notify.VerboseEmail=email
57
58
59
60  notificationSettings =  \
61      factory.EnumerateInstances('Novell_AppArmorNotificationSettingData')[0]
62
63  printNotificationSettings(notificationSettings)
64  changeNotificationsEmail(notificationSettings, 'root@localhost')
65  printNotificationSettings(notificationSettings)
```

Appendix B

Default Logging Configuration

```
[formatters]
keys: detailed,simple

[handlers]
keys: console,file

[loggers]
keys: root,factory,CIMClass

[formatter_simple]
format: %(name)s:%(levelname)s:  %(message)s

[formatter_detailed]
format: %(name)s:%(levelname)s %(module)s:%(lineno)d:  %(message)s

[handler_console]
class: StreamHandler
args: []
formatter: simple

[handler_file]
class: handlers.RotatingFileHandler
args: ['/var/log/powerCIM.log','a']
formatter: detailed

[logger_factory]
level: INFO
handlers: file
qualname=powerCIM.factory

[logger_CIMClass]
```

```
32 level: INFO
33 handlers: file
34 qualname=powerCIM.CIMClass
35
36 [logger_root]
37 level: INFO
38 handlers: console
39 qualname=root
```

Appendix C

IPython configuration

C.1 ipythonrc

```
# -*- Mode: Shell-Script -*-  Not really, but shows comments correctly
#***************************************************************************
#
# Configuration file for ipython -- ipythonrc format
#
# The format of this file is one of 'key value' lines.
# Lines containing only whitespace at the beginning and then a # are ignored
# as comments. But comments can NOT be put on lines with data.
#***************************************************************************

# If this file is found in the user's ~/.ipython directory as
# ipythonrc-powerCIM, it can be loaded by calling passing the '-profile
# powerCIM' (or '-p powerCIM') option to IPython.

# First load basic user configuration
include ipythonrc

# from ... import *
import_all powerCIM

# from ... import ...
import_some

# code
execute factory = WBEMFactory('http://localhost','','')

# Files to execute
execfile powerCIM-magic.py
```

C.2 powerCIM-magic.py

```
import IPython.ipapi

ip = IPython.ipapi.get()

def GetPropertyDocumentation(self, arg):

    ip = self.api

    partition = arg.rpartition('.')
    object = partition[0]
    property = partition[2]

    ip.ex("print %s._GetPropertyDocumentation('%s')" % (object,property))


ip.expose_magic('docu', GetPropertyDocumentation)
```

Appendix D

PowerCIM DVD

A DVD which comes with this work contains a PDF version of this text and source code of the prototype implementation as discussed in Chapter 4. It can be also used as a bootable live DVD to try the prototype capabilities on a running system.

If you want to access the source code and do not want to boot the system, the following directories and files are available on the DVD:

MFF/ is a directory with source code and PDF version of this text. Other directories on this DVD are internal and they are used to make the DVD bootable. They can be safely ignored.

MFF/powerCIM/ contains powerCIM source code as described in Chapter 4.1.

MFF/RPMS/ contains RPM and source RPM of powerCIM packaged for openSUSE Linux distribution[1].

MFF/ScriptingOfCIM.pdf is a pdf version of this work.

This DVD is bootable and you can use it for testing powerCIM capabilities on any x86-based computer[2].

If you want to try this linux distribution follow these steps:

1. Insert the DVD into your DVD drive
2. Boot your computer from your DVD drive

[1] It should be possible to use the source RPM to build powerCIM on other RPM based linux distributions. This was not tested, though.

[2] Please note that although this DVD is based on openSUSE Linux which is tested on many different computers, this DVD was not tested extensively and therefore it is quite likely that it will not run on a given hardware configuration. If this is the case, it is a bug of the distribution and it is not in the scope of this work to fix such issues.

3. Once the system is up and running, you will see a KDE desktop. To run a terminal program, you can press ALT+F2 and type `konsole`.
4. In a terminal you have several options. For example, you can type "`ipython`" to get a system management console based on IPython and powerCIM that can be used to manage the live system.

   You can also use the terminal to examine the directory structure discussed below.

In the home directory, (type "`cd ~`" in `konsole`) the following directories might be of your interest:

powerCIM/examples is a directory with powerCIM usage examples. You can try to run the `guiServices` example discussed in Appendix A.1 using the following commands:

```
1 cd ~/powerCIM/examples
2 ./guiServices.py
```

A window similar to Figure A.1 should appear[3]. Other examples can be executed similarly.

powerCIM/rpms contains rpm (source rpm) used to install (build) powerCIM for this live DVD.

powerCIM/testing contains powerCIM testsuite.

powerCIM/IPython contains files used for IPython integration as discussed in Section 4.6.

[3]Unfortunately some of the start/stop button does not work as one would expect. This is problem of the corresponding CIM providers on the WBEM server side.

List of Figures

1.1 Meta Schema Structure [4] 7
1.2 Operating System Example (MOF) 9
1.3 Operating System Example (UML) 10
1.4 WBEM Components [1] 13
1.5 WBEM Server, Clients and Providers [1] 14

2.1 Example – List Services Using Microsoft VBScript [27] 18
2.2 Example – List Services Using Windows PowerShell [27] 18
2.3 Example – List Services Using pywbem 19
2.4 Example – List Services Using wbemcli/bash 20
2.5 Wbemcli – Enumerate Instance Names 21
2.6 Example – List Services Using CIM Client for Java 22
2.7 Google Trends - Perl, Python, Ruby [26] 25
2.8 Decorator Example 27

3.1 Get-Compile-Store-Load-Develop Cycle 32
3.2 Accessing Property Qualifiers 40
3.3 Incorrect Access to Descriptor Class Docstring 42
3.4 Accessing Docstring of Descriptor Class 43

4.1 PowerCIM Overview 54
4.2 IPython as CIM Enabled Management Console 65

A.1 guiServices.py Screenshot 71
A.2 An Example of apparmor_report.py Output 75

Bibliography

[1] C. Hobbs. *A Practical Approach to WBEM/CIM Management.* Auerbach Publications, 2004.

[2] Forman, Ira R., and Scott Danforth. *Putting Metaclasses to Work.* Addison-Wesley, 1999.

[3] *Cambridge Advanced Learner's Dictionary.* Cambridge University Press, 2005.

[4] Distributed Management Task Force, Inc. *Common Information Model (CIM) Infrastructure Specification.* DSP0004, Version 2.3 Final. 07 August 2007 <http://www.dmtf.org/standards/published_documents/DSP0004V2.3_final.pdf>.

[5] Distributed Management Task Force, Inc. *Specification for CIM Operations over HTTP.* DSP0200, Version 1.2. 07 August 2007 <http://www.dmtf.org/standards/published_documents/DSP200.html>.

[6] Distributed Management Task Force, Inc. *Specification for the Representation of CIM in XML.* DSP0201, Version 2.2. 07 August 2007 <http://www.dmtf.org/standards/published_documents/DSP201.html>.

[7] Distributed Management Task Force, Inc. *Solution Exchange and Service Incident Specification.* DSP0132, Status: Preliminary. 07 August 2007 <http://www.dmtf.org/standards/documents/CIM/DSP0132.pdf>.

[8] Distributed Management Task Force, Inc. *WBEM Discovery Using the Service Location Protocol.* DSP0205, Status: Preliminary. 07 August 2007 <http://www.dmtf.org/standards/documents/CIM/DSP0132.pdf>.

[9] Distributed Management Task Force, Inc. *Web Services for Management (WS-Management).* DSP0226, Status: Preliminary. 07 August 2007 <http://www.dmtf.org/standards/published_documents/DSP0226.pdf>.

[10] OMG Available Specification. *Object Constraint Language, Version 2.0.* 03 August 2007 <http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf>.

[11] "Command line completion." *Wikipedia: The Free Encyclopedia.* 25 July 2007 <http://en.wikipedia.org/wiki/Tab_completion>.

[12] "Simple Network Management Protocol." *Wikipedia: The Free Encyclopedia.* 05 May 2007 <http://en.wikipedia.org/wiki/Simple_Network_Management_Protocol>.

[13] "Windows Management Instrumentation." *Wikipedia: The Free Encyclopedia.* 08 May 2007 <http://en.wikipedia.org/wiki/Windows_Management_Instrumentation>.

[14] "Xen." *Wikipedia: The Free Encyclopedia.* 06 August 2007 <http://en.wikipedia.org/wiki/Xen>.

[15] "CIM Tutorial." *Distributed Management Task Force and WBEM Solutions, Inc.* 05 May 2007 <http://www.wbemsolutions.com/tutorials/CIM/>.

[16] Mertz, David and Michele Simionato. "Metaclass programming in Python." *IBM developerWorks.* 18 April 2007 <http://www-128.ibm.com/developerworks/linux/library/l-pymeta.html>.

[17] Mertz, David and Michele Simionato. "Metaclass programming in Python, Part 2." *IBM developerWorks.* 18 April 2007 <http://www-128.ibm.com/developerworks/library/l-pymeta2/>.

[18] Mertz, David. "Charming Python: Decorators make magic easy." *IBM developerWorks.* 30 April 2007 <http://www-128.ibm.com/developerworks/linux/library/l-cpdecor.html>.

[19] O'Brien, Patrick. "Guide to Python introspection." *IBM developerWorks.* 28 May 2007 <http://www.ibm.com/developerworks/library/l-pyint.html>.

[20] Python Software Foundation. "How do I write a function with output parameters (call by reference)?." *effbot.org!.* 03 August 2007 <http://effbot.org/pyfaq/how-do-i-write-a-function-with-output-parameters-call-by-reference.htm>.

[21] Pérez, Fernando. "IPython – An enhanced Interactive Python – User Manual, v. 0.8.1". *IPython.* 9 May 2007 <http://ipython.scipy.org/doc/manual/>.

[22] Eckel, Bruce. "5-2-03 Strong Typing vs. Strong Testing." *Bruce Eckel's MindView, Inc.* 29 May 2007 <http://www.mindview.net/WebLog/log-0025>.

[23] Eckel, Bruce. "3-31-04 I'm Over It." *Bruce Eckel's MindView, Inc.* 07 August 2007 <http://www.mindview.net/WebLog/log-0053>.

[24] Spolsky, Joel. “Language Wars.” *Joel on Software*. 20 May 2007 <http://www.joelonsoftware.com/items/2006/09/01.html>.

[25] Ascher, David. “Dynamic Languages – ready for the next challenges, by design.” *ActiveState Software, Inc.* 10 June 2007 <http://www.activestate.com/Corporate/Publications/ActiveState_Dynamic_Languages.pdf>.

[26] “About Google Trends.” *Google Trends*. 20 May 2007 <http://www.google.com/intl/en/trends/about.html>.

[27] “Accessing WMI From Windows PowerShell.” *Microsoft TechNet Script Center*. 08 May 2007 <http://www.microsoft.com/technet/scriptcenter/topics/msh/mshandwmi.mspx>.

[28] Rossum, Guido van. “Python Reference Manual, Release 2.5.” *Python Documentation*. 19 September 2006 <http://docs.python.org/ref/ref.html>.

[29] Rossum, Guido van. “Python Tutorial, Release 2.5.” *Python Documentation*. 19 September 2006 <http://docs.python.org/tut/>.

[30] Rossum, Guido van. “Python Library Reference, Release 2.5.” *Python Documentation*. 19 September 2006 <http://docs.python.org/lib/>.

[31] Rossum, Guido van. “What is Python? Executive Summary.” *Python Documentation*. 27 May 2007 <http://www.python.org/doc/essays/blurb/>.

[32] Lemburg, Marc-André. “Attribute Docstrings”. *Python Developers Guide*. 23 August 2000 <http://www.python.org/dev/peps/pep-0224/>.

[33] Smith, Kevin D., Jim J. Jewett, Skip Montanaro, and Anthony Baxter. “Decorators for Functions and Methods.” *Python Developers Guide*. 05 Jun 2003 <http://www.python.org/dev/peps/pep-0318/>.

[34] Kuchling, A. M. “Python Advanced Library.” *Python Developers Guide*. 19 Jun 2007 <http://www.python.org/dev/peps/pep-0206/>.

[35] “AppArmor Application Security for Linux.” *NOVELL: Worldwide*. 03 August 2007 <http://www.novell.com/linux/security/apparmor/>.

[36] Campbell, Leona Beatrice and Jana Jaeger. “Novell AppArmor 2.0 Administration Guide.” *NOVELL: Worldwide*. 04 July 2006 <http://www.novell.com/documentation/suse101/pdfdoc/apparmor-admin-guide_en/apparmor-admin-guide_en.pdf>.

[37] “Wide Adoption for UML Techniques.” *Methods & Tools*. 03 August 2007 <http://www.methodsandtools.com/dynpoll/oldpoll.php?UMLPoll>.