



December 3-6, 2007, Santa Clara Marriott, Santa Clara, CA

The Common Manageability Programming Interface (CMPI)

(including changes for version 2.0)

Dave Sudlik

IBM Poughkeepsie

dsudlik@[us.ibm.com](mailto:dsudlik@us.ibm.com)

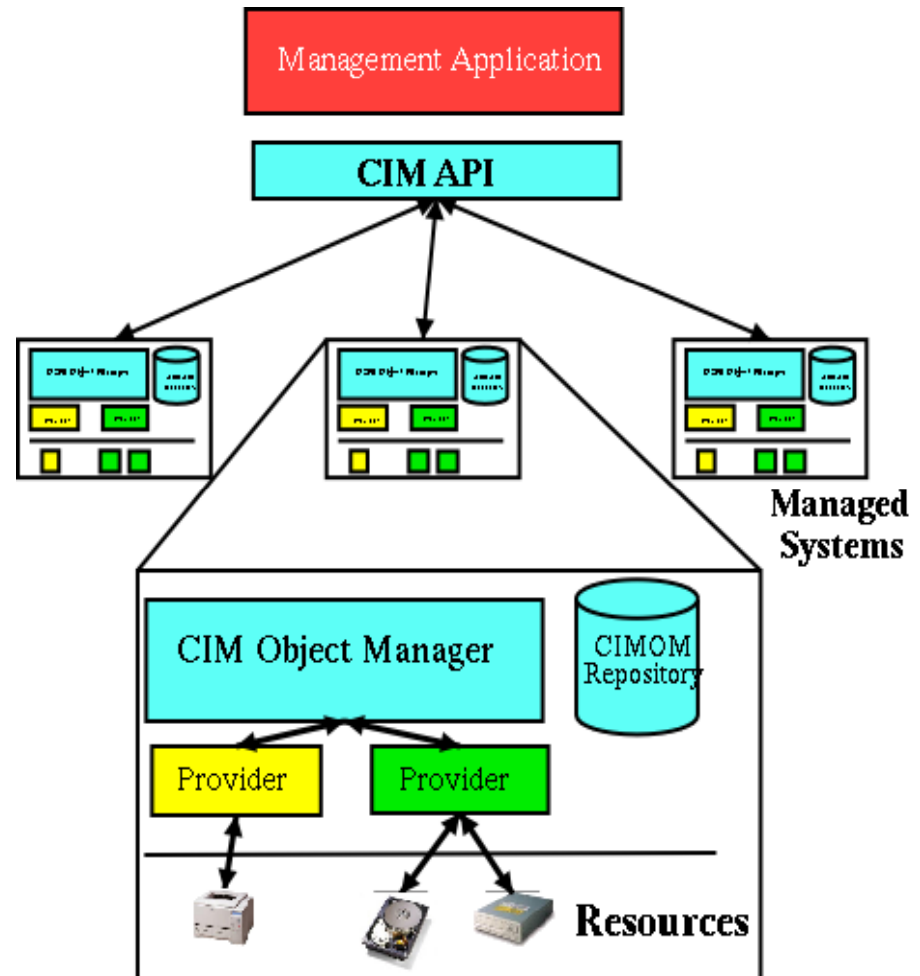


Agenda

- Introduction to Providers
- CMPI Overview
- CMPI Providers
 - Provider Types and Interfaces
 - Data Types
 - Provider Flows
 - CMPI Broker Services
 - Provider Considerations
- Roadmap and References

Provider Concepts

- Providers are the implementation for the standard data model.
- Like “device drivers” they encapsulate IT resources by implementing standard interfaces that allow to manipulate and query those resources through CIM.
- Providers are developed independent from Management Applications, ideally by the same team that develops the resource encapsulated by the provider.
- Management Applications make use of existing providers through the standard CIM interfaces, rather than establishing their own agent technology.





Provider Implementations

- There's a rich variety of APIs to implement CIM providers
 - OpenPegasus C++ API, OpenWBEM C++ API, Java JSR48, CMPI (C / C++ / Perl) , ...
- The good news is that providers have a simple overall structure - to a certain extent.
- It's all about creating and manipulating instances of CIM Classes.
- Their purpose is to convert the native resource data to CIM data types and vice versa.
- Access to the resources is usually encapsulated into a separate module/library

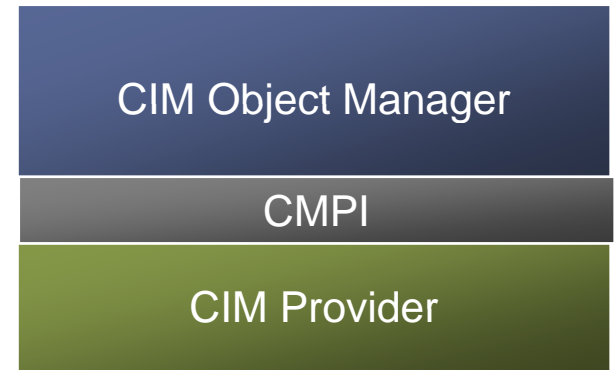
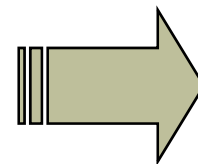


Providers

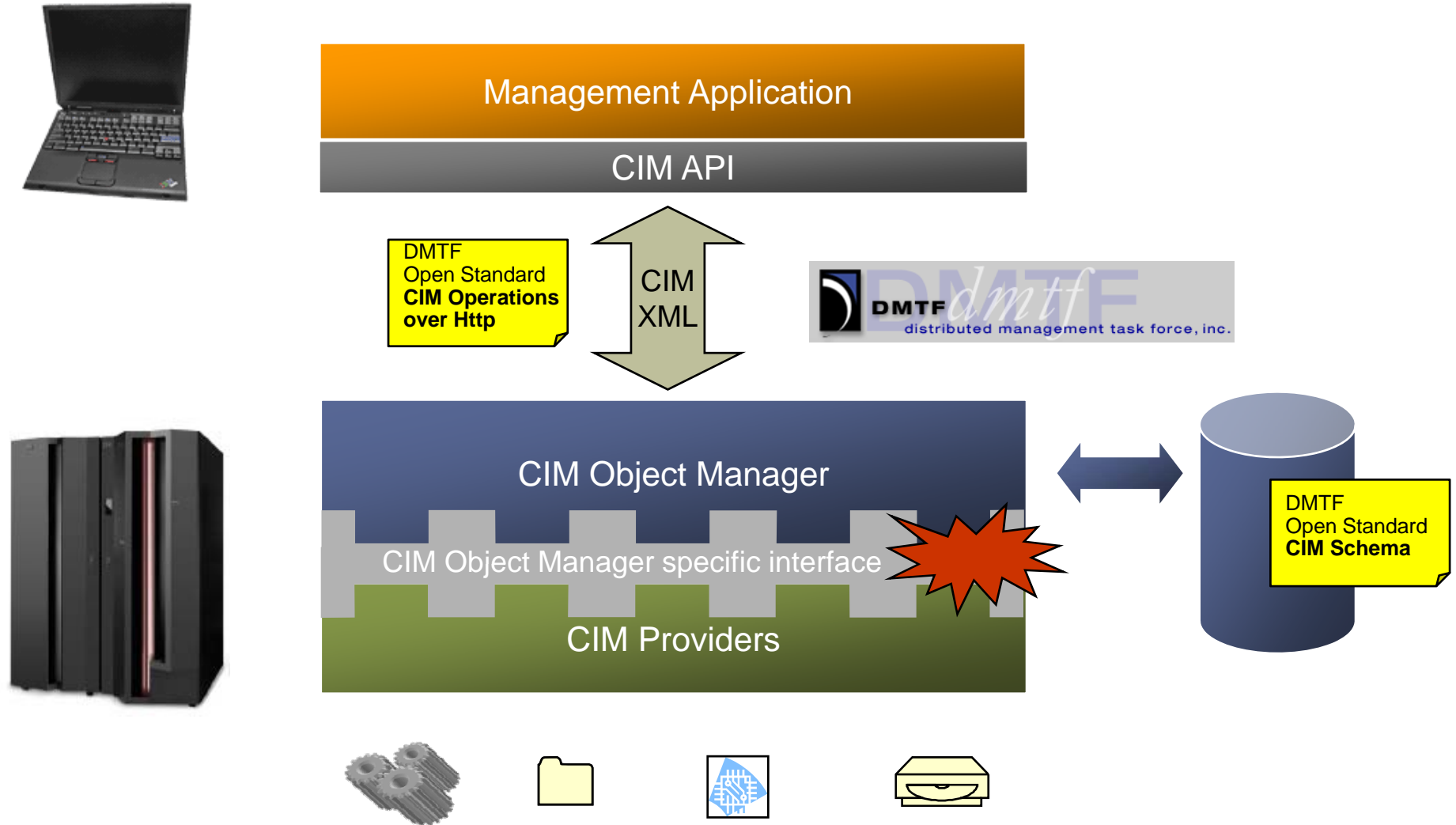
- Providers are “registered” for a specific CIM class and get “called” **ONLY** for that class.
- The provider writer specifies which classes to implement.
- The provider creates the **object(s)** for the **class(es)**.
- Classes are skeletons – they have methods, properties, qualifiers, etc. They are abstractions.
- Objects are the instances of classes – they have “live” data and the logic to process, acquire values, etc.
- Providers are dynamic libraries with functions that correspond to CIM operations.
- They are loaded on demand (when a user requests the operation).

Overview of CMPI

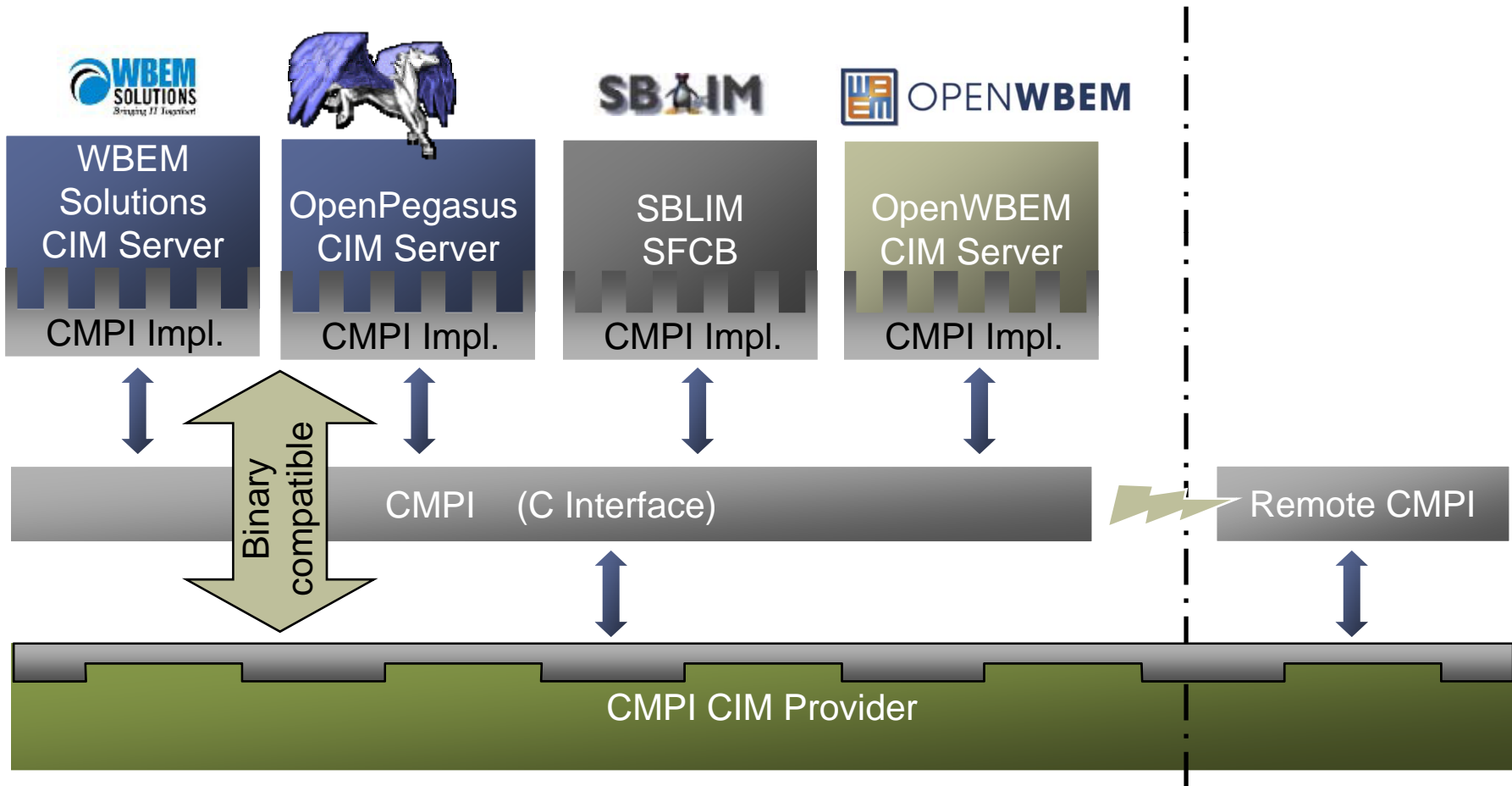
- Common Manageability Programming Interface (CMPI) is a C API for providing resource data for CIM operations.
- It's an Open Group Standard
- Implemented by all open-source WBEM implementations (OpenPegasus, SFCB, OpenWBEM, SNIA) and closed-source as well: WBEM Solutions.
- Its major objective is to provide binary compatibility between different CIM Object Managers.
- Small memory footprint, doesn't require linking against any CIMOM library.



The Provider Interface Problem



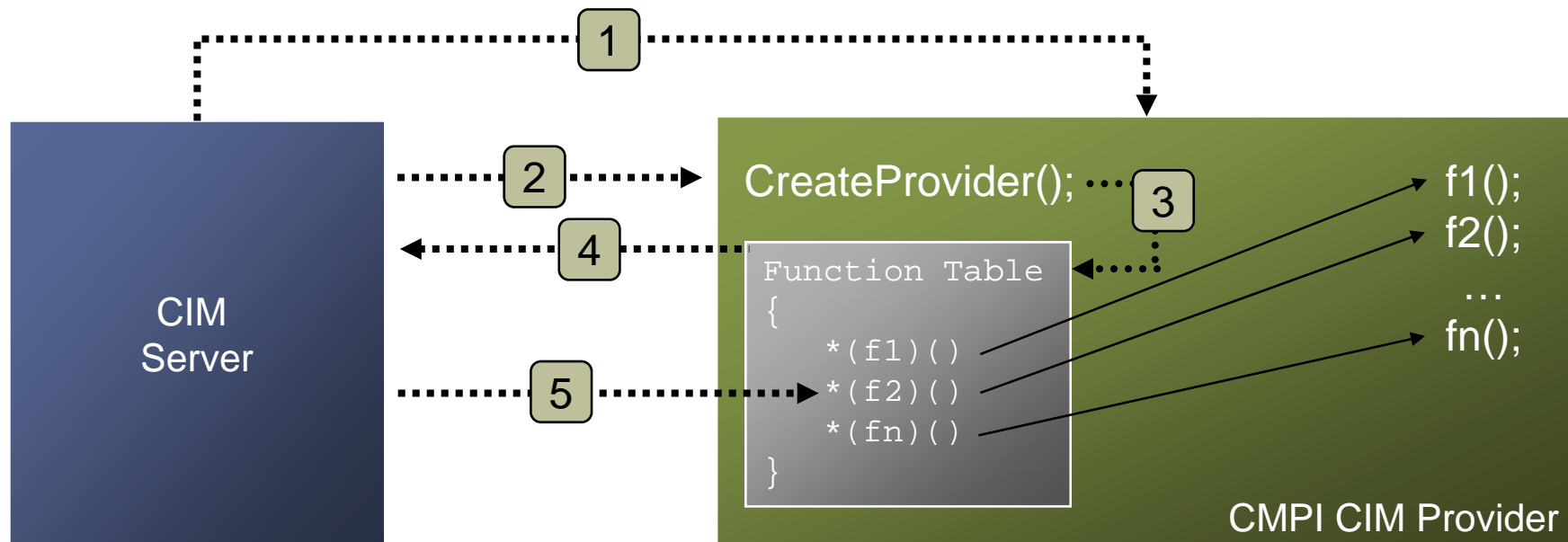
The CMPI Provider Interface



CMPI Provider Interface Technique

1. CIM Object Manager loads each shared library of one or more provider <fix>
2. CIM Object Manager invokes provider's well known Create() function.
3. The Create() function initializes the provider's function table with function pointers to the implemented provider functions.
4. Provider's function table is returned to CIM Object Manager
5. CIM Object Manager invokes provider function through pointer from function table.

→ **No static linking occurs between CIM Object Manager and Provider!**





The CMPI Provider API for C

- **The CMPI Provider API for C is an Open Group standard**
 - The CMPI version 2.0 specification is available on the web at:
<http://www.opengroup.org/bookstore/catalog/c061.htm>
- **Header files are provided as part of OpenPegasus source code:**
 - CMPI function prototypes:
pegasus/src/Pegasus/Provider/CMPI/cmpif.h
 - CMPI data types:
pegasus/src/Pegasus/Provider/CMPI/cmpidt.h
 - CMPI convenience macros:
pegasus/src/Pegasus/Provider/CMPI/cmpimacs.h

Or by download from the Open Group CMPI website:

 - <http://www.opengroup.org/tech/management/cmpi/>
- **The SBLIM project provides a general provider architecture document for CMPI Providers:**
 - <http://sblim.sourceforge.net/doc/ProviderArchitecture.pdf>

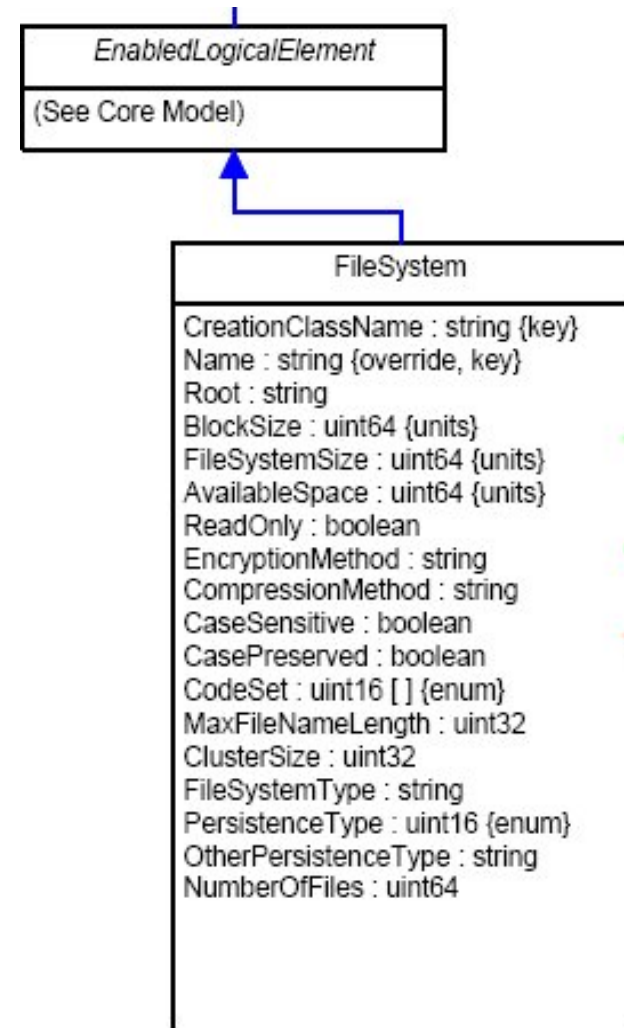


Types of CMPI Providers

- **Instance**
 - Retrieving and manipulating of instances of CIM classes.
- **Method**
 - Executing methods of CIM classes on object instances
- **Association**
 - Implementation of relationship between objects
- **Property**
 - Retrieving and manipulating of a single property of an object instance
- **Indication (Events)**
 - Registration, filtering and raising of events

Instance Providers

- Instance Providers are the most common kind of Management Instrumentation. They allow access to resources incorporating the principal elements of a model
- The operations “**enumerate instances**”, “**enumerate instance names**” and “**get instance**” are essential for resource access and should be considered mandatory.
- The operations “**create instance**”, “**modify instance**” and “**delete instance**” must be implemented if the model usage mandates that instance management is done via explicit creation, modification or deletion of elements.
- The operation “**execute query**” should always be implemented, if the cost of enumeration is considered too high





CMPI Instance Provider Interface

```
CMPI Status cleanup( CMPI InstanceMI *, const CMPI Context *, CMPI Boolean term);
```

```
CMPI Status enumerateInstanceNames( CMPI InstanceMI *, const CMPI Context *,  
                                     const CMPI Result *, const CMPI ObjectPath *);
```

```
CMPI Status enumerateInstances( CMPI InstanceMI *, const CMPI Context *,  
                                 const CMPI Result *, const CMPI ObjectPath *, const char **properties);
```

```
CMPI Status getInstance( CMPI InstanceMI *, const CMPI Context *, const CMPI Result *,  
                          const CMPI ObjectPath *, const char** properties);
```

```
CMPI Status createInstance( CMPI InstanceMI *, const CMPI Context *, const CMPI Result *,  
                             const CMPI ObjectPath *, const CMPI Instance *);
```

```
CMPI Status modifyInstance( CMPI InstanceMI *, const CMPI Context *, const CMPI Result *,  
                             const CMPI ObjectPath *, const CMPI Instance *,  
                             const char **properties);
```

```
CMPI Status deleteInstance( CMPI InstanceMI *, const CMPI Context *,  
                              const CMPI Result *, const CMPI ObjectPath* );
```

```
CMPI Status execQuery( CMPI InstanceMI *, const CMPI Context *, const CMPI Result *,  
                       CMPI ObjectPath *, const char *lang, const char *query);
```



Property Providers

- Property providers may be implemented when the cost of retrieving a single instance property is significantly lower than the cost of retrieving the entire instance.
- Property providers can be responsible for one or more properties of an instance. The operations are "get property" and "set property".
- Property providers are NOT necessary in order to support the CIM Get/SetProperty Operations. A CIM Object Manager is using the Instance provider for properties as well by default.



CMPI Property Provider Interface

```
CMPI Status cleanup( CMPIPropertyMI *mi , const CMPIContext* ctx, CMPI Boolean term);
```

```
CMPI Status setProperty ( CMPIPropertyMI *mi , const CMPIContext *ctx,  
    CMPI Result *result, CMPI ObjectPath *op,  
    const char *name, const CMPI Data data);
```

```
CMPI Status getProperty ( CMPIPropertyMI * mi , const CMPIContext *ctx,  
    CMPI Result *result, CMPI ObjectPath *op,  
    const char *name);
```

Method Providers

- Method Providers are needed to implement model instance methods.
- The only operation is "invoke method". A single method provider can be used for one or more methods of an instance.





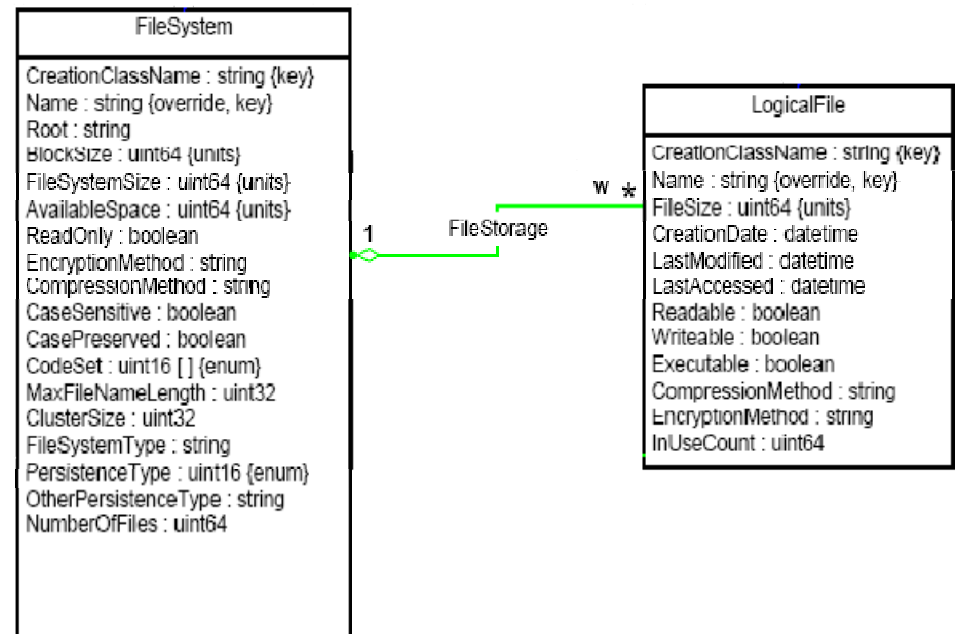
CMPI Method Provider Interface

```
CMPIStatus cleanup( CMPIMethodMI * mi, const CMPIContext * ctx, Boolean term);
```

```
CMPIStatus invokeMethod( CMPIMethodMI * mi, const CMPIContext * ctx,  
    const CMPIResult * rslt, const CMPIObjectPath * op,  
    const char * methodName, const CMPIArgs * in,  
    CMPIArgs * out);
```

Association Providers

- Association Providers are needed for navigation between the principal elements, they are usually implemented in conjunction with (at least one of) the associated Instance providers.
- The operations "**associators**" and "**associator names**" are used to retrieve elements that are connected to a given element. They return - so to speak – the nodes.
- The operations "**references**" and "**reference names**" return the association instances between elements, or the edges. These operations are less commonly used but should be implemented as well.





CMPI Association Provider Interface

```
CMPI Status Cleanup( CMPI AssociationMI * mi, const CMPI Context * ctx, Boolean term);
```

```
CMPI Status Associators( CMPI AssociationMI * mi, const CMPI Context * ctx,  
    const CMPI Result * rslt, const CMPI ObjectPath * op,  
    const char * assocClass, const char * resultClass,  
    const char * role, const char * resultRole,  
    const char ** propertyList );
```

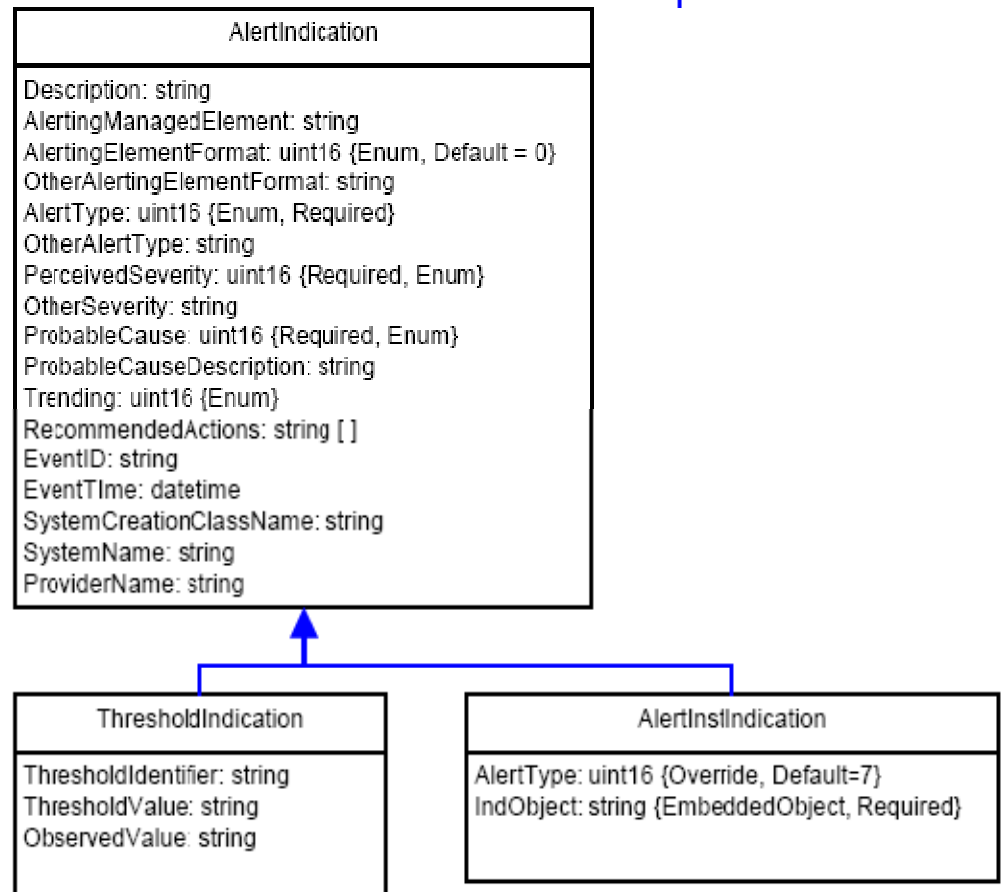
```
CMPI Status AssociatorNames( CMPI AssociationMI * mi, const CMPI Context * ctx,  
    const CMPI Result * rslt, const CMPI ObjectPath * op,  
    const char * assocClass, const char * resultClass,  
    const char * role, const char * resultRole);
```

```
CMPI Status References( CMPI AssociationMI * mi, const CMPI Context * ctx,  
    const CMPI Result * rslt, const CMPI ObjectPath * op,  
    const char * assocClass, const char * role,  
    const char ** propertyList );
```

```
CMPI Status ReferenceNames( CMPI AssociationMI * mi, const CMPI Context * ctx,  
    const CMPI Result * rslt, const CMPI ObjectPath * op,  
    const char * assocClass, const char * role);
```

Indication Providers

- Event or Indication Providers must be implemented for event subscription and notification.
- They must implement the operations "authorize filter", "activate filter" and "deactivate filter"
- Usually, event providers are delivering indications asynchronously to a subscriber. For this they need to call the broker function `deliverIndication()`.
- In order to work efficiently, an event provider should itself register to some kind of notification mechanism to avoid overhead caused by polling





CMPI Indication Provider Interface

```
CMPI Status Cleanup( CMPI IndicationMI *, const CMPI Context* ctx, Boolean term);
```

```
CMPI Status AuthorizeFilter( CMPI IndicationMI *, const CMPI Context * ctx,  
    const CMPI SelectExp * filter, const char * classname,  
    const CMPI ObjectPath * co, const char * owner );
```

```
CMPI Status MustPoll ( CMPI IndicationMI *, const CMPI Context * ctx,  
    const CMPI Result * rslt, const CMPI SelectExp * filter,  
    const char * classname, const CMPI ObjectPath * co );
```

```
CMPI Status ActivateFilter( CMPI IndicationMI *, const CMPI Context * ctx,  
    const CMPI SelectExp * filter, const char * classname,  
    const CMPI ObjectPath * co, CMPI Boolean first );
```

```
CMPI Status DeActivateFilter( CMPI IndicationMI *, const CMPI Context * ctx,  
    const CMPI SelectExp * filter, const char * classname,  
    const CMPI ObjectPath * co, CMPI Boolean last );
```

```
CMPI Status EnableIndications( CMPI IndicationMI *, const CMPI Context * ctx );
```

```
CMPI Status DisableIndications( CMPI IndicationMI *, const CMPI Context * ctx );
```

Basic CMPI Data Types

- CMPIChar16
- CMPIUInt8
- CMPIUInt16
- CMPIUInt32
- CMPIUInt64
- CMPI Sint8
- CMPI Sint16
- CMPI Sint32
- CMPI Sint64
- CMPIReal32
- CMPIReal64
- CMPIString
(Zero-terminated UTF-8 char array)
- CMPIDateTime
- CMPIInstance
- CMPIObjectPath
- CMPIArgs
- CMPISelectExp
- CMPIEnumeration
- CMPIArray
- CMPIValuePtr
- ...

Returning Data

GetInstance, EnumerateInstances, Associators, References:

```
CMPI Status CMPIResultFT.returnInstance( const CMPIResult* rslt,  
                                             const CMPIInstance* inst );  
... or CMReturnInstance(...)
```

EnumerateInstanceNames, AssociatorNames, ReferenceNames:

```
CMPI Status CMPIResultFT.returnObjectPath( const CMPIResult* rslt,  
                                             const CMPIObjectPath* ref );  
... or CMReturnObjectPath(...)
```

Asynchronous Indication delivery:

```
CMPI Status CMPIBrokerFT.deliverIndication( const CMPIBroker* mb,  
                                             const CMPIContext* ctx,  
                                             const char* ns, );  
... or CBDeliverIndication(...)
```

Returning Instances

- **Construct Class Object Path for Instance**
 - Classname must be the returned instance's class name
 - Use namespace from passed-in object path
 - `CMNewObjectPath()`
- **Construct Instance for Class Object Path**
 - `CMNewInstance()` ←
- **Add Properties**
 - Must pass pointer to data and data type (see `cmpidt.h`)
 - `CreationClassName` = instance class name
 - `CM SetProperty()`
- **Return Instance(s)**
 - `CMReturnInstance()`
- **Close Result Handler**
 - `CMReturnDone()`



Returning Instance Object Paths

- **Construct Object Path for Instance**
 - Classname must be the returned instance's class name
 - Use namespace from passed-in object path
 - `CMNewObjectPath()` ←
- **Add Keys**
 - Must pass pointer to data and data type (see `cmpidt.h`)
 - `CreationClassName` = instance class name
 - `CMAddKey()`
- **Return Object Path**
 - `CMReturnObjectPath()`
- **Close Result Handler**
 - `CMReturnDone()`

Provider Initialization Flow

- CIM Object Manager is initially calling the MI factory function `<provider name>_Create_<MI>MI`.
- where `<MI>` can be one of Instance, Association, Method, Property or Indication.
- The factory function returns a pointer to `<MI>MI` with the MI function table set up to point to the MI functions.
- The convenience macro `CM<MI>MIStub` can be used for simple providers. (CMI nstanceMI Stub, ...)



Provider Initialization

```

CMI nstanceMI Stub( My_Operati ngSystemProvi der, // provider name used in code
                   My_Operati ngSystemProvi der, // provider name used for registrati on
                   _broker,
                   my_i ni t_routi ne()); // custom ini ti al izati on routi ne

```

Extends to ...

```

#define CMI nstanceMI Stub(pfx, pn, broker, hook) \
static CMPI InstanceMI FT i nstMI FT__={ \
    CMPI CurrentVersi on, \
    CMPI CurrentVersi on, \
    "i nstance" #pn, \
    pfx##Cl eanup, \
    pfx##EnumI nstanceNames, \
    pfx##EnumI nstances, \
    pfx##GetI nstance, \
    pfx##Createl nstance, \
    pfx##SetI nstance, \
    pfx##Del etel nstance, \
    pfx##ExecQuery, \
}; \
CMPI _EXTERN_C \
CMPI InstanceMI * pn##_Create_I nstanceMI (CMPI Broker* brkr, CMPI Context *ctx) { \
    static CMPI InstanceMI mi={ \
        NULL, \
        &i nstMI FT__, \
    }; \
    broker=brkr; \
    hook; \
    return &mi; \
}

```



Actual names of the implemented provider functions.



EnumerateInstances Flow

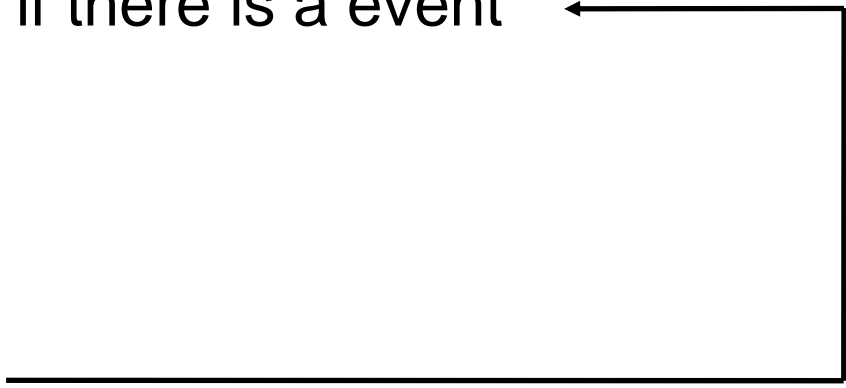
- **<ProviderName>_Create_InstanceMI()** – called only first time when loading provider. This function returns the MI's function table.
- **EnumerateInstances()** from the MI's function table is then executed.
- In EnumerateInstances function:
 - **CMNewObjectPath()** is called to create a new object path.
 - **CMNewInstance()** is called to create an instance (or more than one)
- Start retrieving the resource data
 - **CMSetProperty()** for each property
 - **CMReturnInstance()** for each instance that is finished
- **CMReturnDone()** when completed.



InvokeMethod Flow

- Confirm that the classname is correct
 - `CMGetClassName()`
 - `CMGetCharsPtr()`
- Confirm that the method is correct
 - Check `methodName`
- Extract argument values if method supports it
 - `CMGetArgCount()`
 - `CMGetArgAt()` or `CMGetArg()`
- Return data via output argument, if method supports it
 - `CMAAddArg()`
- If method returns data via return value, do that:
 - `CMReturnData()`
 - `CMReturnDone()`

Indication Flow

- In AuthorizeFilter check that the classname extract from the FROM clause is correct
 - In ActivateFilter:
 - CBPrepareAttachThread()
 - Create new thread.
 - In the thread:
 - CBAttachThread()
 - Monitor the data in a loop, and if there is a event 
 - CMNewObjectPath()
 - CMNewInstance()
 - CMSetProperty()
 - CBDeliverIndication()
- Keep cycling in the loop
- CBDetachThread()



Common Parameters

- Nearly all provider functions are called with some common parameters
- **CMPI <MI >MI *** - Pointer to MI structure as generated by factory function - not needed, unless you want to do some extra fancy stuff
- **CMPI Context*** - Invocation Context – used for thread creations and can contain information about the requestor (client) ID and language settings.
- **CMPI Result*** - Result Handler - used to return instances, object paths or data from provider functions
- **CMPI ObjectPath*** - Target object path for request - used to extract namespace, classname and (optionally) key values

Tracing and Logging

- It is recommended to use the CMPI logging and tracing features :
 - `CMLogMessage`(int severity, const char* id, const char* text, const CMPIString *string);
 - `CMTraceMessage`(int level, const char* component, const char* text, const CMPIString *string);



Management Broker Services

(upcalls)

Capability	Dependency	CMPIBrokerFT Functions
Basic Read	None	GetInstance EnumerateInstances EnumerateInstanceNames GetProperty
Basic Write	Basic Read	SetProperty
Instance Manipulation	Basic Write	CreateInstance ModifyInstance DeleteInstance InvokeMethod
Association Traversal	Basic Read	Associators AssociatorNames References ReferenceNames
Query Execution	Basic Read	ExecQuery
Indications	Basic Read	DeliverIndication



Threading Services

(available for any CMPI provider types)

- **attachThread()**
 - Inform the CMPI run-time system that the current thread with the specified context will begin using CMPI services.
- **prepareAttachThread()**
 - Prepare the CMPI run-time system to accept a thread that will be using CMPI services.
- **detachThread()**
 - Inform the CMPI run-time system that the current thread will no longer be using CMPI services.
- **exitThread()**
 - Causes the current thread to exit with the passed in return code.



OS Abstraction Services

- CMPI defines OS abstraction services for the following areas:
 - Thread handling
 - Semaphore handling
 - Library name resolution
- These services are optional to be implemented by a CIM Object Manager.
- Availability is indicated by the **CMPI_MB_OSEncapsulationSupport** flag in the `brokerCapabilities`.



A simple Example 1 of 2

```
CMPI Status My_OperatingSystemProviderGetInstance( CMPI InstanceMI * mi ,
                                                    const CMPI Context * ctx,
                                                    const CMPI Result * rslt,
                                                    const CMPI ObjectPath * op,
                                                    const char ** properties ) {

    CMPI Instance * ci = NULL;
    CMPI Status rc = {CMPI_RC_OK, NULL};

    "DoSomeChecking()"

    /* Call into ObjectFactory */
    ci = _makeInst( _broker, ctx, op, &rc );

    /* Return result back to CIMOM */
    CMReturnInstance( rslt, ci );           // only 1 instance
    CMReturnDone( rslt );                 // tell callback we're done

    return rc;                            // tell broker result
}
```



A simple Example 2 of 2

```
CMPI Instance * _makeInst( const CMPI Broker * _broker,
                           const CMPI Context * ctx,
                           const CMPI ObjectPath * ref,
                           CMPI Status * rc
                           ) {

    CMPI ObjectPath          * op   = NULL;
    CMPI Instance           * ci   = NULL;
    struct cim_operatingsystem * sptr = NULL;
    int                      frc   = 0;

    /* Create new object path for object instance */
    op = CMNewObjectPath( _broker, CMGetCharPtr(CMGetNamespace(ref, rc)),
                          OSCreationClassName, rc );

    /* Create a new object instance */
    ci = CMNewInstance( _broker, op, rc);

    /* Call into resource access layer */
    frc = get_operatingsystem_data(&sptr);

    /* Add properties to object instance */
    CMSetProperty( ci, "NumberOfUsers", (CMPI Value*)&(sptr->numOfUsers), CMPI_uint32);
    CMSetProperty( ci, "NumberOfProcesses", (CMPI Value*)&(sptr->numOfProcesses), CMPI_uint32);

    return ci;
}
```



Provider Considerations

- Every element constructed and returned by an provider **has to reside in a namespace**. Use the namespace from the passed-in object path for this purpose.
- Providers can (and will) be called in a **polymorphic** manner. I.e. "enumerate instances" could be requested for a ManagedElement. The passed-in object path will therefore have the classname "CIM_ManagedElement" and cannot be used to construct instances of the proper class. Providers need therefore to be aware of the class names they are responsible for.
- Do not use **writable global values**, as they might be shared during **multi threaded execution** of the providers. If you have to, use mutexes to protect them.



Provider Considerations

- Instance, Property and Method MIs are only called by the broker if they are responsible for a given class. This is not necessarily true for Association MIs. Association MIs must therefore check the classname of the passed-in object path and return immediately for classes they are not providing.
- Do not de-allocate CMPI structures as they are **garbage collected** at the execution finish. This is true only for CMPI structures that have not been cloned.
- Do your own memory management of your data structures (note that in CMPI v2.0 there will be means of the broker doing that for you).
- Use the CMPI mutex/conditionals/thread functionality instead of linking against pthread.



Outlook & Roadmap

- Corrigendum for CMPI 2.0 is expected to be released by year-end 2007
 - Look for it on the Open Group bookstore
- Next revision of CMPI (2.1?) is being defined by an Open Group workgroup
 - C++ interface definitions
 - Pulled enumerations (WIP CR000386)
 - more...
- Proposals and implementations for wrapping CMPI providers with C++
 - <http://sblim.sourceforge.net/doc/CMPIandC++.pdf>
 - OpenPegasus provides “experimental” C++ CMPI provider interfaces (not part of standard yet)



Related Projects & Organizations

- SBLIM – many examples of CMPI providers.
 - <http://sourceforge.net/projects/sblim/>
- OpenPegasus
 - <http://www.openpegasus.org/>
- DMTF
 - <http://www.dmtf.org/home>
- Open Group
 - <http://www.opengroup.org/management/>

Summary

- CMPI is a **vital standard**
- CMPI provides **loose coupling and binary compatibility** between Provider and CIM Object Manager.
- CMPI is **widely used** and supported.



Sample Code Copyright Disclaimer

The following statement applies to all sample/pseudo code in this presentation:

```
* @copyright (di scl ai mer) *
* Licensed Materials - Property of IBM *
* Product name *
* (c ) Copyright IBM Corp. 1998, 2005. All Rights Reserved. *
* *
* US Government Users Restricted Rights *
* Use, duplication or disclosure restricted by GSA ADP Schedule *
* Contract with IBM Corp. *
* *
* DI SCLAI MER OF WARRANTI ES : *
* *
* Permission is granted to copy and modify this Sample code, and to *
* distribute modified versions provided that both the copyright *
* notice, - and this permission notice and warranty disclaimer appear in all copies *
* and modified versions. *
* *
* THIS SAMPLE CODE IS LICENSED TO YOU AS-IS. *
* IBM AND ITS SUPPLIERS AND LICENSORS DI SCLAI M ALL WARRANTI ES, EI THER EXPRESS OR *
* IMPLIED, IN SUCH SAMPLE CODE, I NCLUDING THE WARRANTI Y OF NON-I NFRI NGEMENT AND THE *
* IMPLIED WARRANTI ES OF MERCHANTABI LITY OR FI TNESS FOR A PARTI CULAR PURPOSE. I N NO *
* EVENT WI LL I BM OR I TS LICENSORS OR SUPPLIERS BE LI ABLE FOR ANY DAMAGES ARI SI NG OUT *
* OF THE USE OF OR I NABI LITY TO USE THE SAMPLE CODE, DI STRI BUTI ON OF THE SAMPLE CODE, *
* OR COMBI NATION OF THE SAMPLE CODE WI TH ANY OTHER CODE. I N NO EVENT SHAL L I BM OR I TS *
* LICENSORS AND SUPPLIERS BE LI ABLE FOR ANY LOST REVENUE, LOST PROFIT S OR DATA, OR *
* FOR DI RECT, I NDI RECT, SPECI AL, CONSEQUENTI AL, I NCI DENTAL OR PUNI TI VE DAMAGES, *
* HOWEVE R CAUSED AND REGARDLESS OF THE THEORY OF LI ABILI TY, -, EVEN I F I BM OR I TS *
* LICENSORS OR SUPPLIERS HAVE BEEN ADVI SED OF THE POSSI BI LI TY OF SUCH DAMAGES. *
* *
* @endCopyri ght
```