

November 15-18, 2010



Santa Clara Marriott
Santa Clara, CA

The Common Manageability Programming Interface (CMPI)

Karl Schopmeyer
President, Inova Development Inc.
k.schopmeyer@swbell.net
k.schopmeyer@opengroup.org

THE *Open* GROUP
Making standards work®

Agenda

- Short Overview of Providers
- CMPI Overview
- CMPI Providers - Overview
- Roadmap
- References
- Some alternative ideas – Making CMPI simpler

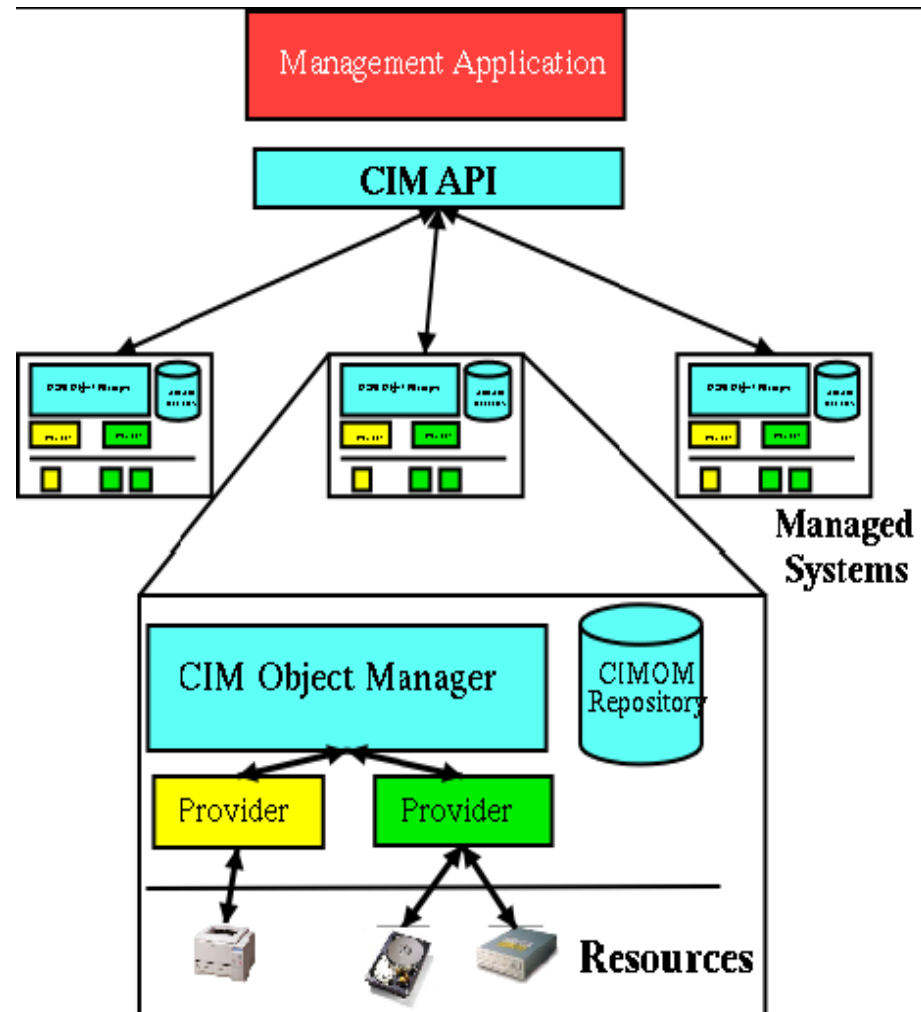
CMPI Terminology

- CMPI Introduces implementation neutral terminology

CMPI	Implementations
Management Broker (MB or Broker)	CIM Object Manager, CIM Server, etc.
Management Instrumentation (MI)	Provider, CIM Provider, Resource Manager

Provider Concepts

- Implementation for the standard data model
- Like “device drivers” the encapsulate IT resources by implementing standard interfaces that allow manipulation and query of those resources through the model
- Providers are developed independent of Management applications
- Management Applications make use of the model through providers and the standard WBEM client interfaces rather than establishing their own infrastructure



Provider Implementations

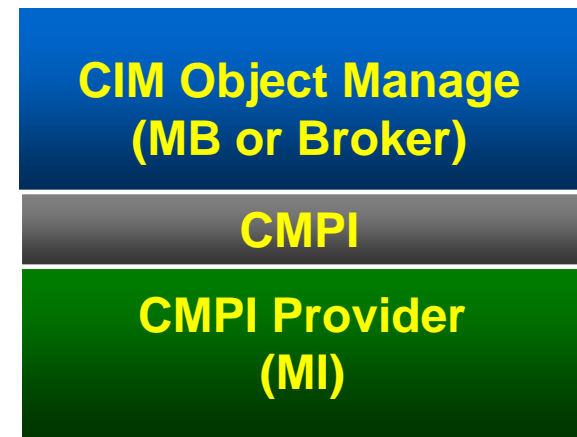
- Variety of APIs to implement CIM Providers
 - Standards based - JSR48, CMPI
 - Other Interfaces - OpenPegasus C++ API, OpenWbem C++ API, etc.
 - Providers have a simple overall structure.
 - All about:
 - creating and manipulating instances of the CIM model
 - » Remember, associations are just instances
 - Generating indications based on subscriptions
 - Purpose is to map native resource data and relationships to the CIM Model and vice versa
 - Access to the resources is usually encapsulated into a separate module/library

Providers

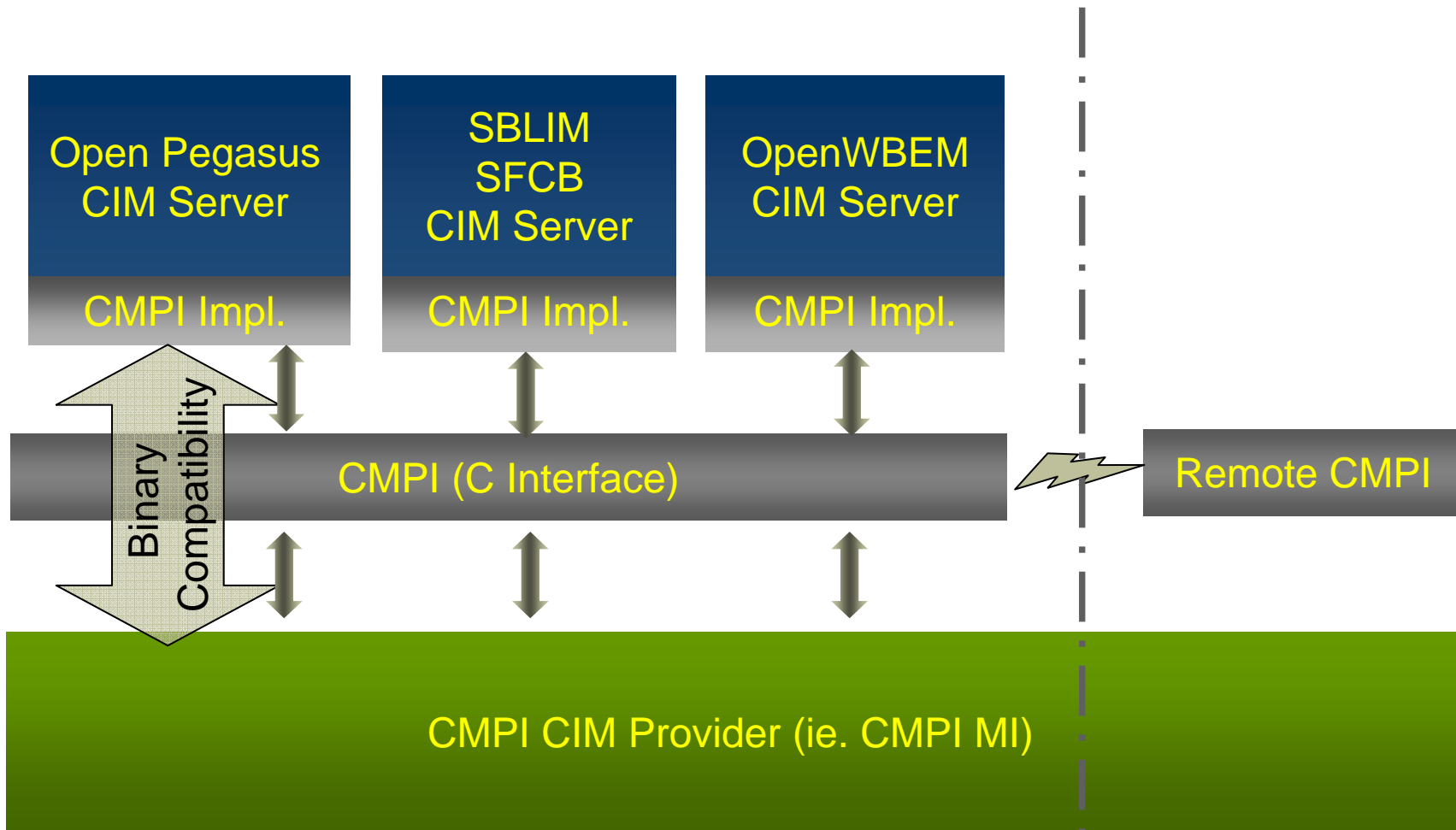
- Providers **register** with MB for a specific CIM class and called **ONLY** for that class
- Provider writer specifies classes to implement
- The provider creates the **object(s)** for the **class(es)**
- **Classes are skeletons** – they define properties, methods, qualifiers, etc. They are **abstractions**
- Instances of classes contain live data and logic to process, acquire values, etc.
- Providers are **dynamic libraries** with functions which correspond to WBEM operations
- Providers normally **loaded on demand** (user requests an operation for the class)

CMPI Overview

- **CMPI is a C API** for providing resource data for CIM Operations
- **CMPI is an OpenGroup** standard
 - Currently version 2
- Implemented today by all open-source WBEM CIMServer implementations
 - OpenPegasus, OpenWBEM, SFCB
- Implemented by a number of closed source implementations
 - WBEMSolutions servers
- Major objective is to provide compatibility between different CIM Object Managers (MBs)
- **Small footprint**, doesn't require linking against CIMOM library

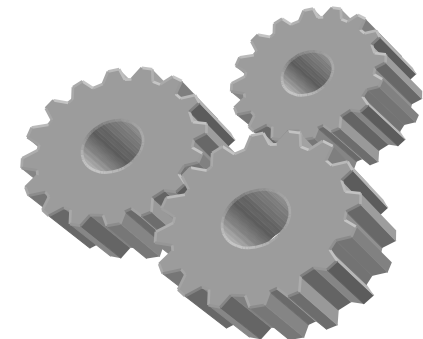


CMPI MI(Provider) Interface



CMPI Specification

- CMPI Specification is an OpenGroup standard available on the web at:
 - <http://www.opengroup.org/bookstore/catalog/c061.htm> (version 2.0)
 - <http://www.opengroup.org/bookstore/catalog/c051.htm> (version 1.0)
- Header files are available from multiple sources
 - CIM Object Manager distributions
 - Ex.OpenPegasus source code
 - Platform-neutral copies of these header files are available at:
 - <http://www.opengroup.org/tech/management/cmapi/>
- CMPI Header files
 - `cmPIDt.h` – CMPI data types (CMPI_VERSION = 100,200)
 - `cmPIft.h` – CMPI function prototypes
 - `cmPImacs.h` – CMPI convenience macros
 - `cmPIos.h` – Operating system specific definitions
 - `cmPIpl.h` – CMPI platform definitions
- Other Documentation
 - SBLIM Architecture Document
 - <http://sblim.sourceforge.net/doc/ProviderArchitecture.pdf>



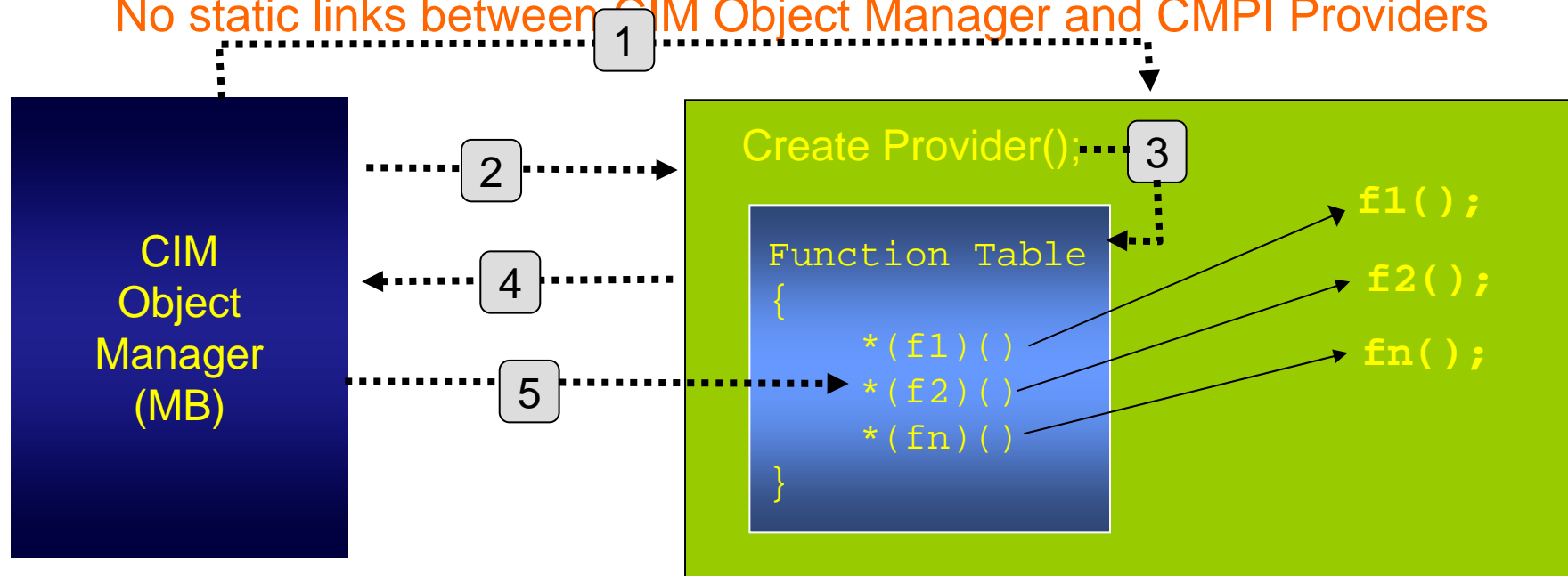
Changes for V2 CMPI

- Standardized Header file names
 - `cmpidt.h`, `cmpift.h`, `cmpios.h` `cmpipl.h`
- Standardized CMPI macro header file `cmpimacs.h`
- Enhanced messaging API
- `setPropertyWithOrigin()`. Allow providers to set origin on instance property
- `CMPIInstanceFT::setObjectPath()` may be used to set keys of instance
- Clarification of filter parsing functions
- Introduce `CMPIError`, etc.
- `CMPIBrokerExtFT` (os system abstraction interface) required
- Introduce `CMPIBrokerMemFT` (memory functions)

CMPI MI Interface Technique

1. MB loads library containing MI functions
2. MB invokes MI's well-known Create() function
3. Create() functions initializes the MI's function table with function pointers to the implemented MI functions
4. MI's function table is returned to MB
5. MB invokes MI functions through pointer from function table

No static links between CIM Object Manager and CMPI Providers

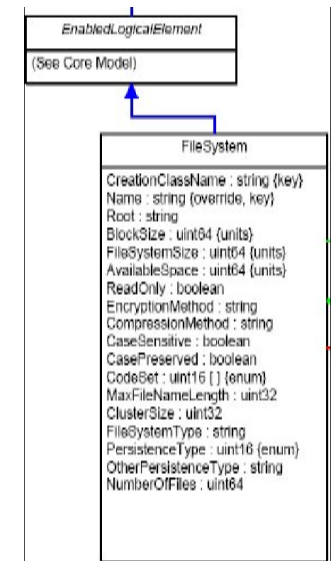


CMPI MI Types

- **Instance**
 - Retrieving and manipulating instances of CIM Classes using the equivalent of CIM Intrinsic methods
- **Association**
 - Define relationships between CIM Objects. Add Intrinsic methods for traversing relationships
- **Method**
 - Execute extrinsic methods of CIM Classes on object instance (or the class (static))
- **Property**
 - Retrieve and set single property of an object instance
 - NOTE: This is being deprecated in DMTF
- **Indication**
 - Subscription registration, filtering and sending of CIM Indications

Instance Providers (MIs)

- Most common form of Management Instrumentation
- Allow access to resources incorporating the principal elements of a model
- Read operations (normally mandatory)
 - Enumerate instances
 - Enumerate Instance Names
 - Get Instance
- Modification operations (model manipulation)
 - Create instance
 - Modify Instance
 - Delete Instance
 - Use if model/profile dictates that instance management is done via explicit creation, modification or deletion of elements
- **exec query** – Useful for providers delivering large quantities of instances. You should implement this function



CMPI Instance MI Interface

- Defined in CMPIInstanceMIFT struct
 - cleanup
 - createInstance
 - deleteInstance
 - getInstance
 - enumerateInstancenames
 - enumerateInstances
 - execQuery

```
static CMPIInstanceMIFT
    instMIFT =
{
    0,
    1,

    "FooCorp_ComputerSystem",
    instCleanup,
    enumInstanceNames,
    enumInstances,
    getInstance,
    createInstance,
    setInstance,
    deleteInstance,
    execQuery
};
```



CMPI Instance MI Interface Prototypes

```
CMPIStatus cleanup( CMPIInstanceMI*, const CMPIContext*, CMPIBoolean term);
```

```
CMPIStatus enumerateInstanceNames(CMPIInstanceMI*, const CMPIContext*, const  
    CMPIResult*, const CMPIObjectPath*, const char** properties);
```

```
CMPIStatus enumerateInstances(CMPIInstanceMI*, const CMPIContext*, const  
    CMPIResult*, const CMPIObjectPath*, const char** properties);
```

```
CMPIStatus getInstance(CMPIInstanceMI*, const CMPIContext*, const  
    CMPIResult*, const CMPIObjectPath*, const char** properties);
```

```
CMPIStatus createInstance (CMPIInstanceMI*, const CMPIContext*, const  
    CMPIResult*, const CMPIObjectPath*, const CMPIInstance*);
```

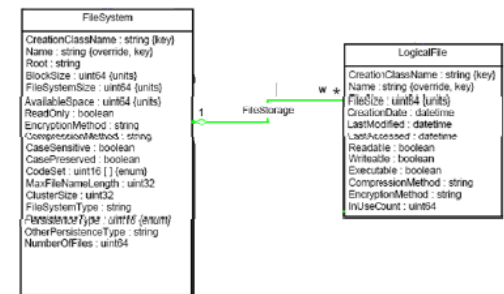
```
CMPIStatus modifyInstance(enumerateInstances(CMPIInstanceMI*, const  
    CMPIContext*, const CMPIResult*, const CMPIObjectPath*, const CMPIInstance  
    * const char** properties )
```

```
CMPIStatus deleteInstance(enumerateInstances(CMPIInstanceMI*, const  
    CMPIContext*, const CMPIResult*, const CMPIObjectPath*);
```

```
CMPIStatus execQuery(enumerateInstances(CMPIInstanceMI*, const CMPIContext*,  
    const CMPIResult*, const CMPIObjectPath*, const char *lang, const char*  
    query);
```

Association Providers (MIs)

- Needed for navigation between the principal elements. Usually implemented with (at least one of) the associated Instance MIs
- **associators** and **associator names** operations retrieve elements that are associated to a given element.
- **references** and **reference names** return the association instances between elements.
- Normally all should be implemented



CMPI Provider Interfaces

CMPIAssociationMIFT

- Defined in CMPIAssociationMIFT
- Functions
 - cleanup
 - associators
 - associatorNames
 - references
 - referenceNames

```
static CMPIAssociationMIFT
    assocMIFT =
{
    0,
    1,
    "FooCorp_HostedFileSystem",
    assocCleanup,
    associators,
    associatorNames,
    references,
    referenceNames
};
```



CMPI Association MI Interface prototypes

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

```
CMPIStatus associators( CMPIAssociationMI* mi const  
    CMPIContext* ctx, const CMPIResult* rslt, const  
    CMPIObjectPath* op, const char *assocClass, const char*  
    resultClass, const char* role, const char* resultRole, const  
    char** propertyList);
```

```
CMPIStatus associatorNames( CMPIAssociationMI* mi const  
    CMPIContext* ctx, const CMPIResult* rslt, const  
    CMPIObjectPath* op, const char *assocClass, const char*  
    resultClass, const char* role, const char* resultRole);
```

```
CMPIStatus references( CMPIAssociationMI* mi const  
    CMPIContext* ctx, const CMPIResult* rslt, const  
    CMPIObjectPath* op, const char *assocClass, const char*  
    role, const char** propertyList);
```

```
CMPIStatus referenceNames( CMPIAssociationMI* mi const  
    CMPIContext* ctx, const CMPIResult* rslt, const  
    CMPIObjectPath* op, const char *assocClass, const char*  
    role);
```

Method Providers (MIs)

- Implement model instance methods
- The only operation is “**invoke method**”. A single method MI can be used for one or more methods of an instance.

CMPI Method MI Interface

- **Defined in**
CMPIMethodMIFT struct
- **Functions**
 - Cleanup
 - invokeMethod

```
static CMPIMethodMIFT
methodMIFT =
{
    0,
    1,

    "FooCorp_PrintService",
    methodCleanup,
    invokeMethod
};
```



CMPI Method MI Interface prototypes

```
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,  
                         const CMPIArgs* out);
```

Property (MIs)

- May be implemented when cost of retrieving a single instance property is significantly lower than cost of retrieving the entire instance.
- Operations
 - `getproperty`
 - `set property`
- Property MIs are NOT necessary to support CIM Get/SetProperty Operations. An MB may be mapping to the Instance MI (with PropertyList)

CMPI Property MI Interface

- Defined in CMPIPropertyMIFT struct

```
CMPIStatus cleanup( CMPIPropertyMI * mi,  
                   const CMPIContext* ctx,  
                   CMPIBoolean term);
```

```
CMPIStatus SetProperty(CMPIPropertyMI * mi,  
                      const CMPIContext* ctx,  
                      const CMPIResult *rslt,  
                      const CMPIObjectPath * op,  
                      const char * name,  
                      const CMPIData data);
```

```
CMPIStatus SetProperty(CMPIPropertyMI * mi,  
                      const CMPIContext* ctx,  
                      const CMPIResult *rslt,  
                      const CMPIObjectPath * op,  
                      const char * name,  
                      const CMPIData data);
```

Indication MIs

- Indication MIs must be implemented for indication subscription and notification.
- Must implement operations:
 - Authorize filter
 - Activate filter
 - Deactivate filter
 - Usually deliver indications asynchronously to the MB. For this the call the broker function `deliverIndication()`

CMPI Indication MI Interface

- cleanup
- activateFilter
- deactivateFilter
- authorizeFilter
- mustPoll
- enableIndications
- disableIndications

```
static CMPIIndicationMIIFT
    indMIIFT =
{
    0,
    1,
    "FooCorp_FileSystem",
    indCleanup,
    indActivateFilter,
    indAuthorizeFilter,
    mustPoll,
    enableIndications,
    disableIndications
};
```



CMPI Indication MI Interface

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

```
CMPIStatus authorizeFilter( CMPIIndicationMI * mi,  
    const CMPIContext * ctx,  
    const CMPISelectExp * filter,  
    const char * classname,  
    const CMPIObjectPath * co,  
    const char * owner);
```

```
CMPIStatus mustPoll( CMPIIndicationMI * mi,  
    const CMPIContext * ctx,  
    const CMPIResult * rslt  
    const CMPISelectExp * filter,  
    const char * classname,  
    const CMPIObjectPath * co,  
    CMPIBoolean last);
```



CMPI Indication MI Interface

```
CMPIStatus activateFilter( CMPIIndicationMI * mi,
                           const CMPIContext * ctx,
                           const CMPISelectExp * filter,
                           const char * classname,
                           const CMPIObjectPath * co,
                           const char * owner);

CMPIStatus deactivateFilter( CMPIIndicationMI * mi,
                             const CMPIContext * ctx,
                             const CMPIResult * rslt
                             const CMPISelectExp * filter,
                             const char * classname,
                             const CMPIObjectPath * co,
                             CMPIBoolean last);

CMPIStatus enableIndications( CMPIIndicationMI * mi,
                              const CMPIContext * ctx);

CMPIStatus disableIndications( CMPIIndicationMI * mi,
                               const CMPIContext * ctx);
```

Further Explanation

- **activateFilter** - ask the provider to begin monitoring a resource
- **deactiveFilter** - inform the MI that monitoring using this filter should stop
- **authorizeFilter** - ask a provider to verify whether this filter is allowed
- **mustPoll** - ask the MI whether polling mode should be used. MB asking if it should poll by executing `enumerateInstances` at regular interval
- **enableIndications** - tell the MI that indications can now be generated
- **disableIndications** - tell the MI to stop generating
- indications

CMPI

CIM Data Types

CMPI CIM Data Types

- **Simple data types**
 - Map CIM data objects to C data types
- **Encapsulated data types**
 - Map CIM data objects to C structures
- **Query expressions**
 - Map query expression components to C structures

Simple Data Types

CIM Data Type	CMPI Data Type	C Data Type
uint8	CMPIUint8	Unsigned char
sint8	CMPI Sint8	Signed char
uint16	CMPIUint16	Unsigned short
sint16	CMPI Sint16	Short
uint32	CMPIUint32	Unsigned long
sint32	CMPI Sint32	Long
uint64	CMPIUint64	Unsigned long long
sint64	CMPI Sint64	Signed long long
Boolean	CMPI Boolean	Unsigned char
real32	CMPI Real32	Float
real64	CMPI Real64	Double

Encapsulated DataTypes

CIM Object	CMPI Data Type	C Data Structure
Object Path	CMPIObjectPath	struct_CMPIObjectPath
Argument	CMPIArgs	struct_CMPI_Args
Enumeration	CMPIEnumeration	struct_CMPIEnumeration
Array	CMPIArray	Struct_CMPI_Array
Datetime	CMPIdateTime	Struct_CMPIDateTime
String	CMPIString	Struct_CMPIString
Error	CMPIError (2.0)	Struct_CMPIError

Query Data Types

Query Type	CMPI Data Type	C Data Structure
SelectExpression	CMPISelectExp	Struct_CMPISelectExp
Argument	CMPISelectCond	Struct_CMPISelectCond
Enumeration	CMPISubCond	Sturc_CMPI_SubCond
Array	CMPIPredicate	Sturct_CMPIPredicate

“Encapsulated” CMPI Data Types

- Each data type supports lifecycle operations
 - `release()`
 - `clone()`
 - Many `CMPIBrokerEnc` functions are factory services for these data types
- `CMPIResult`
 - `CMPIContext`
 - `CMPIString(/0-term UTF8)`
 - `CMPIArray`
 - `CMPIEnumeration`
 - `CMPIInstance`
 - `CMPIObjectPath`
 - `CMPIArgs`
 - `CMPIDateTime`
 - `CMPISelectExp`
 - `CMPISelectCond`
 - `CMPISubCond`
 - `CMPIPredicate`
 - `CMPIError (2.0)`



CMPIError Data Type (2.0)

- Ability for MI to return detailed error info
 - See CIM_Error in DMTF CIM Schema
 - (Optional) check CMPI_MB_Supports_Extended_Error
- CMPIBrokerEncFT.newCMPIError() factory service
 - Creates new CMPIError object
 - Accepts required CMPIError properties as input
 - Owner, msgID, msg, ErrorSeverity, probableCause, cimStatusCode
- CMPIErrorFT.release()
- CMPIErrorFT.clone()
- CMPIErrorFT.getXXX() – returns CMPIError properties
- CMPIErrorFT.setXXX() – sets optional CMPIError properties
- Where XXX
 - ErrorType, OtherErrorType, OwningEntity, messageID, Message, etc.

Returning Detailed Error Information with Macros

- **Construct CMPIError**
 - Specify required properties (defined by CIM_Error)
 - **CMNewCMPIError()** ←
- **Add desired optional properties**
 - Must pass pointer to CMPIError
 - **CMSet<property>()**
- **Specify detailed Error Information to return**
 - **CMReturnError()** _____
- **Return from MI with rc indicating error (ie. *not CMPI_RC_OK)**
 - **CMReturn()**
 - **CMReturnWithString()**
 - **CMReturnWithChars()**

Returning Data

GetInstance, EnumerateInstances, Associators, References:

```
CMPIStatus CMPIResultFt.returnInstance( const CMPIResult* rslt,  
                                         const CMPIInstance* inst);  
... or CMReturnInstance(...)
```

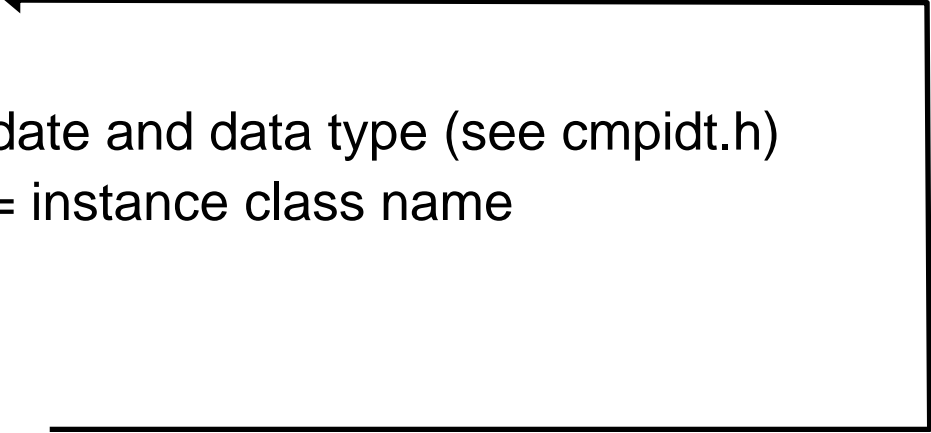
EnumerateInstanceNames, AssociatorNames, ReferenceNames:

```
CMPIStatus CMPIResultFt.returnObjectPath( const CMPIResult* rslt,  
                                           const CMPIObjectPath* inst);  
... or CMReturnObjectPath(...)
```

Asynchronous Indication Delivery:

```
CMPIStatus CMPIResultFt.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 date and data type (see cmpidt.h)
 - CreationClassName = instance class name
 - **CMSetProperty()**
 - **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 (cmpidt.h)
 - CreationClassName = instance class name
 - **CMAddKey()**
- **Return Object Path**
 - **CMReturnObjectPath()**
- **Close Result Handler**
 - **CMReturnDone()**

MI Initialization Flow


- CIM Object Manager (broker) initially calls MI factory function:
 - `<ProviderName>) Create_<MI>MI`
- where `<MI>` can be one of Instance, Association, Method, Property, or Indication
- Factory function returns pointer to `<MI>MI` with MI function table set up to point to MI functions
- Convenience macro `CM<MI>MIStub` can be used for simple MIs

MI Initialization

```
CMInstanceMIStub( My_OperatingSystemProvider, // MI name used in code
                  My_OperatingSystemProvider, // MI name used for registration
                  _broker,
                  my_init_routine());          // custom initialization routine
```

Extends to ...

```
#define CMInstanceMIStub(pfx,pn,broker,hook) \
    static CMPIInstanceMIFT instMIFT__={ \
        CMPICurrentVersion, \
        CMPICurrentVersion, \
        "instance" #pn, \
        pfx##Cleanup, \
        pfx##EnumerateInstanceNames, \
        pfx##EnumerateInstances, \
        pfx##GetInstance, \
        pfx##CreateInstance, \
        pfx##SetInstance, \
        pfx##DeleteInstance, \
        pfx##ExecQuery, \
    }; \
    CMPI_EXTERN_C \
    CMPIInstanceMI* pn##_Create_InstanceMI ( CMPIBroker* brkr,CMPIContext *ctx) { \
        static CMPIInstanceMI mi={ \
            NULL, \
            &instMIFT__, \
        }; \
        broker=brkr; \
        hook; \
        Return \
    }
```



Actual Names of the
implemented MI
Functions

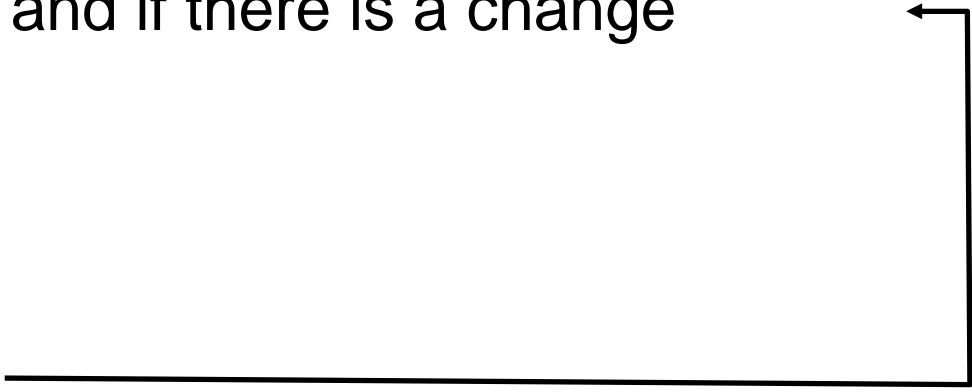
EnumerateInstances Flow

- **<ProviderName>_Create_InstanceMI** – called only the first time when loading MI. This function returns the MI's function table
- **EnumerateInstances** from the MI's function table is then executed
- In EnumerateInstance function:
 - **CIMNewObjectPath** called to create new object path
 - **CIMNewInstance** called to create one or more instances
- Start retrieving the resource data
 - **CIMSetProperty** for each property
 - **CIMReturnInstance** for each instance that is finished
 - **CMReturnDone** when complete

InvokeMethod Flow

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

Indication Flow

- In **authorizeFilter** check that the classname extracted from the FROM clause is correct
 - In activateFilter:
 - **CBPrepareAttachThread**
 - Create new thread
 - In the created thread
 - **CBAttachThread**
 - Monitor the data in loop and if there is a change
 - **CMNewObjectPath**
 - **CMNewInstance**
 - **CMSetProperty**
 - **CBDeliverIndication**
 - Keep cycling in loop
 - **CBDetachThread**
- 

Common Parameters

- Most MI functions are called with common parameters
 - **CMPI<MI>MI *** - Pointer to MI structure as generated by factor function – not needed, unless you want to do extra fancy things
 - **CMPIContext*** - Invocation Context – used for thread creations and can contain information about the requestor ID and language settings
 - **CMPIAcceptLanguage**, **CMPIContentLanguage**
 - **CMPIResult*** Result Handler – used to return instances, object paths or data from MI functions
 - **CMPIObjectPath*** - Target object path for request – used to extract namespace, classname, and (optionally) key values

Tracing and Logging

- Use CMPI logging and tracing features:
 - `CMPIBrokerEncFT.logMessage(...)`, or ...
 - `CMPILogMessage` (`const CMPIBroker * b,`
`int severity,`
`const char* id,`
`const char* text,`
`const CMPIString* string`)
 - `CMPIBrokerEncFt.trace(...)` or ...
 - `CMPITraceMessage` (`const CMPIBroker * b,`
`int level,`
`const char* component,`
`const char* text,`
`const CMPIString *string`)

Message Support (2.0)

- CMGetMessage() not usable. Use new 2.0 message functions
 - **CMOpenMessageFile**(const CMPIBroker* b,
const char* msgFile,
CMPIMsgFileHandle* hdl)
 - **CMGetMessage2**(const CMPIBroker* b,
const char* msgId,
const CMPIMsgFileHandler hdl)
 - **CMCloseMessageFile**(const CMPIBroker * b,
const CMPIMsgFileHandle hdl)



Broker Services

- TBD

Threading Services

- Use for reliable garbage collection
 - **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 use CMPI services
 - **exitThread**
 - Cause the current thread to exit with the passed in return code

OS Abstraction Services

- CMPI defines OS abstraction services to enhance portability for:
 - Thread handling
 - Semaphore handling
 - Library name resolution
- MUST be implemented by a CMPI 2.0 compliant CIM Object Manager
- Availability can be ensured by the **CMPI_MB_OSEncapsulationSupport** flag in `brokerCapabilities`

Memory Services (2.0)

- CMPI defines memory services for the following areas:
 - Extend CMPI garbage collection to heap storage
 - Mark() and release() for long running threads
 - cmpiMalloc(), cmpiCalloc(), cmpiRealloc(), cmpiStrDup(), cmpiFree()
 - freeInstance(), freeObjectPath(), freeString(), freeArray(), etc.
- These service can be optionally implemented by a CIM Object Manager
- Availability indicated by the **CMPI_MB_Supports_MemEnhancements** flag in brokerCapabilities

A simple Example 1 of 2

```
CMPIStatus My_OperatingSystemProviderGetInstance(  
    CMPIInstanceMI * mi,  
    const CMPIContext * ctx,  
    const CMPIResult * rslt,  
    const CMPIObjectPath * op,  
    const char ** properties)  
{  
    CMPIInstance * ci = NULL;  
    CMPIStatus rc = {CMPI_RC_OK, NULL};  
  
    "DoSomeChecking()"  
  
    /* Call into ObjectFactory */  
    ci = _makeInst( _broker, ctx, op, &rc );  
  
    /* Return result back to CIMOM */  
    CMReturnInstance( rslt, ci );  
    CMReturnDone(rslt);  
    return rc;  
}
```

A simple Example 2 of 2

```

CMPIInstance * _makeInst( const CMPIBroker * _broker,
                          const CMPIContext * ctx,
                          const CMPIObjectPath * ref,
                          CMPIStatus * rc)
{
    CMPIObjectPath          * op = NULL;
    CMPIInstance            * 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. Magic occurs here */
    frc = get_operatingsystem_data(&sptr);

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

    return ci;
}

```

MI Considerations

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

MI Considerations

- Instance, Property and Method MIs are only called by the broker if they are responsible for the 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.
- You do not have (but can) release CMPI structures as they are garbage collected at the **execution finish**. This is true only for CMPI structures that have not been cloned. Cloned data must be released by the provider.
- Do your own memory management of your non-CMPI data structures. Note that in CMPI 2.0 there is optional broker support (CMPIBrokerMemFt)
- Use the CMPI mutex/conditionals/thread functionality instead of pthreads

What Else Do I need to Know

- This was an overview. There are many CMPI subjects that we have not covered
 - Filter Parsing
 - The memory model
 - Etc.
- The source of wisdom for CMPI is the CMPI specification. If you use CMPI read the specification
- Look at working examples
 - ex. Pegasus has a number of Sample and test provders
 - SBLIM has a many complete providers



Related Projects and Organizations

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

CMPI Outlook and Roadmap

- Work started on next revision of CMPI (2.x)
 - Goals – Short term 2.x to correct limitations and catch up with other specifications
 - Ideas
 - Expand CIM_Error support (i.e multiple objects, internationalization)
 - Pull Enumerations to the Provider
 - Expand Thread support (permanent threads)
 - Generic Operations semantics support
 - Clarify Embedded Instance handling
 - Utility functions (Ex. Object comparison)
 - Give providers control on server startup and stop
 - Resync error codes with specs
 - Standard operation error message support per generic operations
 - More configuration information available to the provider(ex. Hostname)
- Participation in the OpenGroup CMPI WG is encouraged (or look for next company review ...)

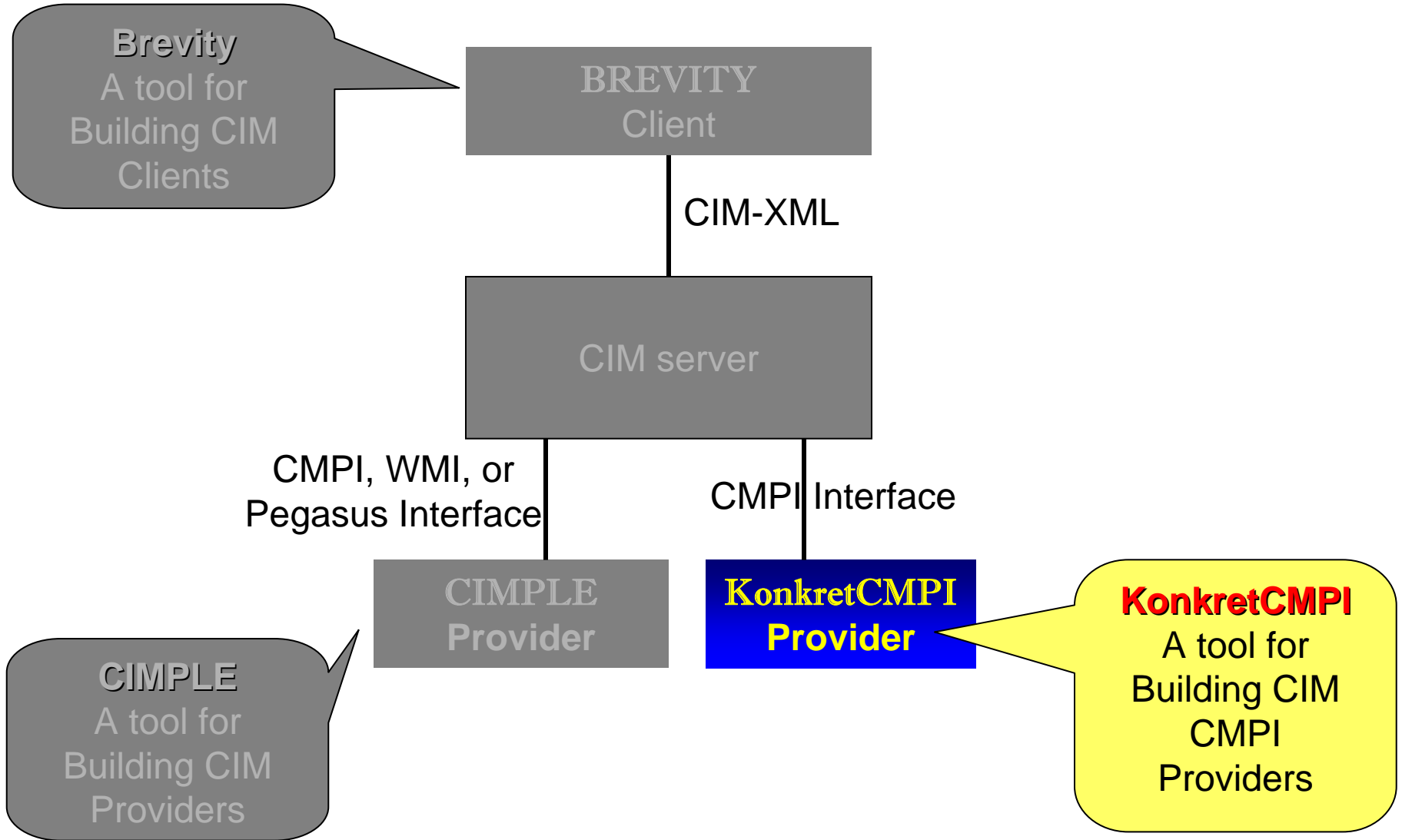
Summary

- CMPI is a vital standard
 - There are only 2 standards based Provider interfaces today
 - CMPI (C)
 - JSR48 (Java)
- CMPI provides loose coupling and binary compatibility between Provider and CMPI Object Manager
- CMPI is widely use and supported

Helping the CMPI Provider writer

- CMPI is heavy on the coding
- Are there alternatives to help writing providers
 - Generating the instance definition from MOF
 - Generating the basic infrastructure for the provider
- A provider generator tool significantly:
 - Improves provider quality
 - Reduces development time
 - Reduces coding and design errors.

KonkretCMPI



What is KonkretCMPI

- **KonkretCMPI** is an open-source environment for developing C CMPI providers.
- **KonkretCMPI** developed providers are truly CMPI providers using the CMPI philosophy
- **KonkretCMPI** provides:
 - Generates concrete C objects from MOF
 - complete default implementations for many provider operations
 - Generates CMPI provider skeleton from MOF
 - Instance skeletons AND method skeletons
 - Builds on CMPI specification
 - Imposes no runtime dependency
 - Produces small footprint providers
 - Provides CMPI convenience functions

Why Use Konkret?

- Reduces provider development effort significantly
- Improves type-safety
 - Generates concrete C objects from MOF
- Provides complete default implementations for many provider operations
- Generates CMPI provider skeleton from MOF
 - Instance skeletons AND method skeletons
- Builds on CMPI specification
- Imposes no runtime dependency
- Produces small footprint providers
- Provides CMPI convenience functions

Different ways to use Konkret

- Use convenience functions
 - Use default operation functions
 - Let KonkretCMPI implement the provider operations
 - Generate class interfaces
 - Generate concrete class interfaces from MOF
 - Generate Provider skeleton
 - Generate complete provider skeleton from MOF
- You do not have to start over to start using KonkretCMPI.

Convenience Functions

- Examples
 - Printing objects
 - Mapping between CMPI objects and Konkret objects
 - Extracting information from CMPI objects
 - ...

Default Provider Operations

- CMPI provides default operations for:
 - getInstance
 - EnumerateInstanceNames
 - Associators
 - AssociatorNames
 - References
 - Referencenames
- You write EnumerateInstances
 - The defaults handle everything else

Generate Class Interfaces

- konkret utility generates class interfaces from class MOF.

```
Class Widget  
{  
    string Key;  
    string Color;  
    unit32 Size;  
}
```

```
konkret -m widget.mof widget
```

Widget.h - C object
representing Class
Widget interfaces

This header can be used to form Widget instances and object paths

Example, forming an instance

- Build instance objects
- Set properties directly
- Map to CMPI instance and object paths

```
// build instance
Const CMPIBroker* _broker
Widget w;
CMPIInstance* ci;
CMPIInstance* cop;
CMPIStatus st;
Widget_Init(&w, _broker,
           KNameSpace(cop));
Widget_Set_Id(&w, "1001");
Widget_Set_Color(&w, "Red");
Widget_Set_Size(&w, 1);

// map to CMPI instance and objectpath
ci = Widget_ToInstance(&w, &st);
cop = Widget_ToObjectPath(&w, &st)
```

Manipulating Properties

- Konkret properties have state (exists, null)
- Konkret creates manipulation functions for each property
 - (<Class>_<function>_<property name>)
- Konkret properties can be
 - **Set** `Widget_Set_Color(&w, "Red");`
 - **Cleared** `Widget_Clr_Key(&w);`
 - **Set to Null** `Widget_Null_Size(&w)`
- Properties can be displayed directly
 - `printf("%s\n", w.Color.chars());`

Mapping to CMPI

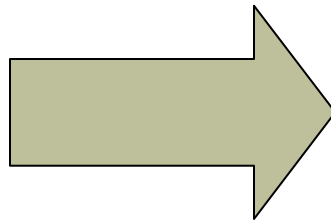
- Instances can be converted to CMPI Instances
 - CMPIInstance* instance;
 - CMPIStatus* status;
 -
 - instance = WidgetToInstance(&w, &status);

Generate Provider Skeletons

- Create a complete Provider skeleton from MOF

konkret -s Widget -m widget.mof widget

```
Class Widget  
{  
    string Key;  
    string Color;  
    unit32 Size;  
}
```



Creates

- Widget.h – Class Widget interfaces
- WidgetProvider.c – Widget provider skeleton

Contains skeletons
for all operations and
methods

Implement EnumerateInstances

- Create Instances and return
- Default skeleton returns zero instances

Your code

The native cmpi Equivalent is about 100 lines of code

```
CMPIStatus WidgetEnumInstances(  
    CMPIInstanceMI* mi,  
    const CMPIContext* context,  
    const CMPIResult* result,  
    const CMPIObjectPath* cop,  
    const char** properties)  
{  
    Widget_Init(&w, _broker KNameSpace(cop));  
    Widget_Set_Key(&w, "1001");  
    KReturnInstance(result,w);  
  
    ...  
    CMReturn( CMPI_RC_OK);  
}
```

Using a default Operation

- Default EnumerateInstanceNames
- Default generated EnumerateInstanceNames uses enumerateInstances

```
CMPIStatus WidgetEnumInstanceNames(  
    CMPIInstanceMI* mi,  
    const CMPIContext* context,  
    const CMPIResult* result,  
    const CMPIObjectPath* cop)  
{  
    return KDefaultEnumerateInstanceNames(  
        _broker, mi, context, result, cop);  
}
```

Association Provider

- Konkret creates class definitions and skeletons
- Minimal implementation is `enumerateInstances`

```
konkret -s Gadget -m widget.mof -m gadget.mof
```

```
[Association] class KC_Gadget  
{  
    [Key] KC_Widget REF Left;  
    [Key] KC_Widget REF Right;  
};
```

Generates

- `Widget.h` – Widget provider class interface
- `Gadget.h` – Gadget Assoc provider class interface
- `GadgetProvider.c` – Gadget provider skeleton

Association Provider

- Implement only `enumerateInstances`
- Create instances with `WidgetRef` function
 - Path subset of `Widget`
- Default associator, reference, ... functions use `EnumerateInstances`
- Developer can add more efficient associators, etc.

```
CMPIStatus GadgetEnumInstances(  
    CMPIInstanceMI* mi,  
    const CMPIContext* cc,  
    const CMPIResult* result,  
    const CMPIObjectPath* cop,  
    const char** properties)  
{  
    const char* ns = KNameSpace(cop);  
    WidgetRef left; WidgetRef right;  
    Gadget g;  
  
    WidgetRef_Init(&left, _broker, ns);  
    WidgetRef_Set_Id(&left, "1001");  
    WidgetRef_Init(&right, _broker, ns);  
    WidgetRef_Set_Id(&right, "1002");  
    Gadget_Init(&g, _broker, ns);  
    Gadget_Set_Left(&g, &left);  
    Gadget_Set_Right(&g, &right);  
    KReturnInstance(result, g);  
    CMReturn(CMPI_RC_OK);  
}
```

Method Provider

- konkret generates skeleton for c function to implement methods defined in class

```
class KC_Widget
{
    [Key] uint32 Id;
    string Color;
    uint32 Size;
    [Static] uint32 Add([In]
        uint32 X,
        [In] uint32 Y);
};
```

- Generator creates Widget_add method
- Default implementation returns NOT_SUPPORTED

```
KUint32 Widget_Add(
    const CMPIBroker* cb,
    CMPIMethodMI* mi,
    const CMPIContext* context,
    const KUint32* X,
    const KUint32* Y,
    CMPIStatus* status)
{
    KUint32 result = KUINT32_INIT;
    KSetStatus(status, ERR_NOT_SUPPORTED);
    return result;
}
```

Implemented Add method

- Developer implements parameter and result setting

```
KUint32 Widget_Add(  
    const CMPIBroker* cb,  
    CMPIMethodMI* mi,  
    const CMPIContext* context,  
    const KUint32* X,  
    const KUint32* Y,  
    CMPIStatus* status)  
{  
    KUint32 result = KUINT32_INIT;  
    if (!X->exists || !Y->exists || X->null || Y->null)  
    {  
        KSetStatus(status, ERR_INVALID_PARAMETER);  
        return result;  
    }  
    KUint32_Set(&result, X->value + Y->value);  
    KSetStatus(status, OK);  
    return result;  
}
```



Indication Provider

- Implemented but I did not have time to document example

Registering Providers

- Konkretreg
 - Builds registration definition from provider library
 - Options for Pegasus (mof files) and sfcb(keyword=value) registration file
 - Different than CIMPLE in that it only creates registration files

Installing KonkretCMPI

- Download source tree
- Untar
- Configure (./configure with schema and CMPI header file options set)
- Make
- Make install (root privileges)

NOTE: KonkretCMPI uses the standard automake utilities

Conclusion

- Konkret is an effective developer tool that matches CMPI philosophy and mechanisms
- Significantly simplifies CMPI provider development
 - Automatically creates provider skeletons
 - Automatically creates class objects
 - Automatically creates methods for creating and manipulating instances, paths, etc.
 - Provides significant compile time error checking
 - At least 10 – 1 source code reduction

Usage

- To date konkret providers have been used on
 - Pegasus cimserver
 - SFCB cimserver
 - (that we know of)

Questions & Discussion



We would like to get your feedback on issues, priorities, users/usage, etc..