

November 15-18, 2010



Santa Clara Marriott
Santa Clara, CA

CIMPLE, BREVITY and KonkretCMPI

Karl Schopmeyer

k.schopmeyer@inovadevelopment.com

V 1;.0 17 Nov 2010

Inova

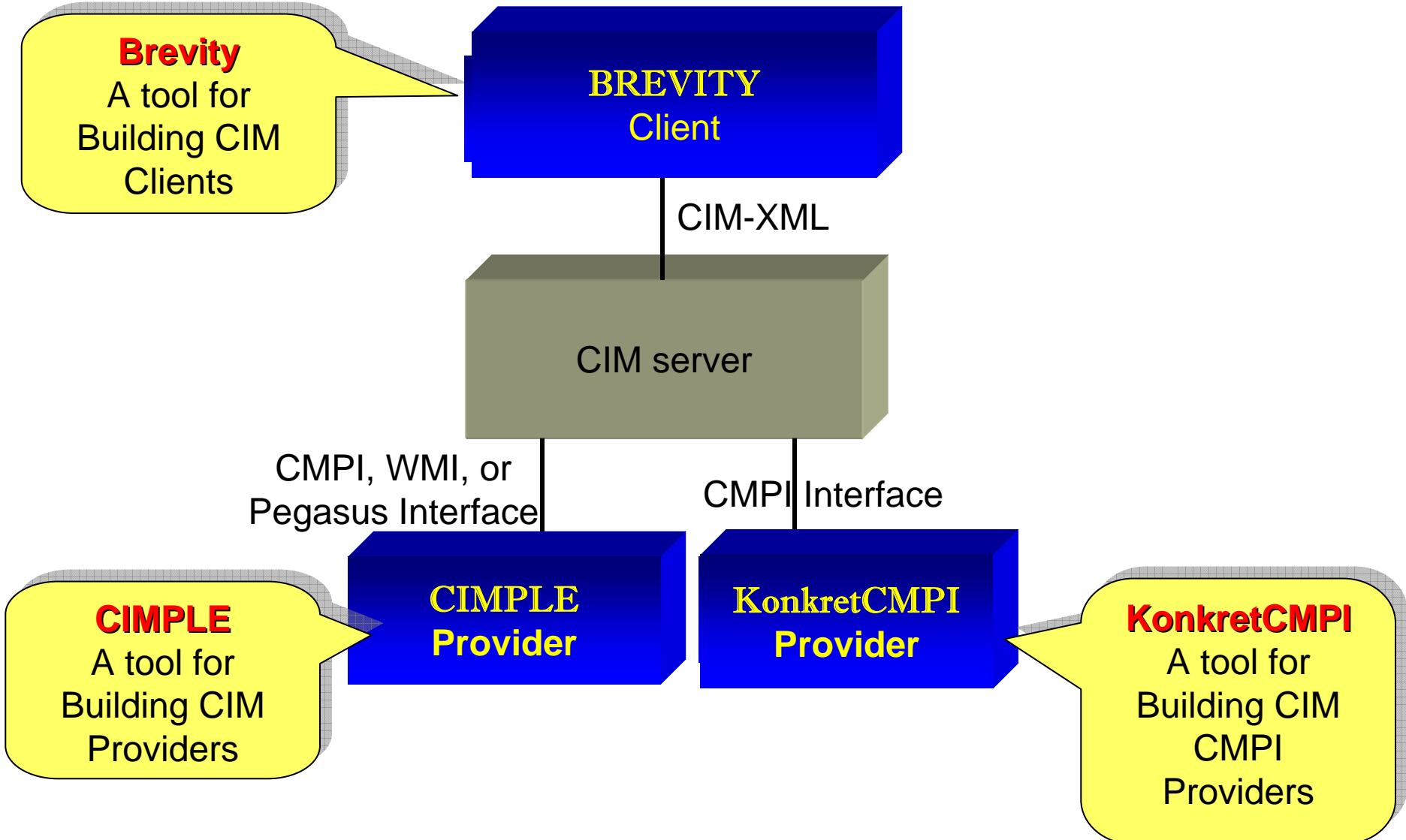
Development



Agenda

- I. Introduction
- II. CIMPLE
- III. BREVITY
- IV. KonkretCMPI
- V. Work in Progress
- VI. Questions

What are CIMPLE, BREVITY and KonkretCMPI?





Provider Development Issues

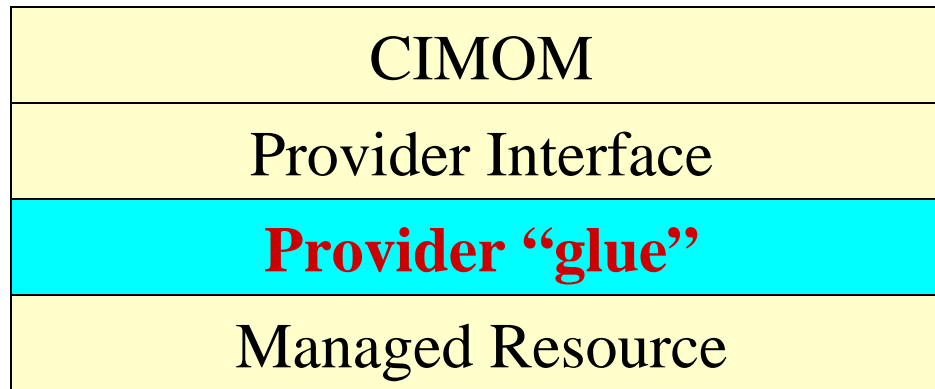
- Conventional providers are:
 - **Difficult** to develop and maintain.
 - **Error prone** – errors caught at run-time rather than compile-time.
 - too **large** for some environments (embedded).
 - too **slow** for some hardware (embedded).
 - **Time consuming to develop** - A typical environment involves many providers
- Management of the provider infrastructure dominates the provider development effort.



Some Core Provider Writing Concepts

- Providers are the heart of a CIM Server. The server itself is just infrastructure support.
- Implementing Profiles not just Providers
- Most providers in a profile are
 - Low usage
 - Standard or simple information
- A few providers in a profile may be high use and/or high volume. These are the ones that make or break the development.
- Provider writing should be a system development activity, not just a coding activity
- Provider development should consider lifecycle, not just coding.
- The core of many providers is
 - Defining and using caching
 - Separating sources of data for efficiency
 - Interface to the information source (often high impedance)

Conventional Provider



The Glue

- Implements cim operations
- Common functions
- Common infrastructure
- Object building/mgt.
- etc.

The provider is largely “glue” that maps a managed resource to a provider interface. **CIMPLE** eliminates the need to develop much of this provider glue.

Issue 1: “dynamic” interfaces

```
UInt32 GetSpeed(CIMInstance& fan) throw(Exception)
{
    UInt32 pos = fan.findProperty("speed");

    if (pos == PEG_NOT_FOUND)
        throw Exception("not found");

    try
    {
        UInt32 speed;
        fan.getProperty(pos).getValue().get(x);
        return speed;
    }
    catch(...)
    {
        throw Exception("type mismatch");
    }
}
```

Dynamic find
and test for
existence of
properties

Dynamic set
value could
cause runtime
exception

What if Fan were a real class?

- Model Properties become variables in the programming language
- No need for much of dynamic checking (existence of properties, values, cimtypes, etc.)
 - Type checking is a compile time issue (Uint32 is a Uint32 variable)
 - One line of code access properties in the concrete object

```
inline Uint32 GetCount(const Fan& fan)
{
    return fan.Speed;
}

// Or just 'fan.Speed'
```



Pegasus/CIMPLE source comparison

```
void Pegasus_example()
{
    try
    {
        CIMInstance inst("TheClass");
        inst.addProperty(CIMProperty("u",
            "hello"));
        inst.addProperty(CIMProperty("v",
            Uint32(99)));
        inst.addProperty(CIMProperty("w",
            Boolean(true)));
        inst.addProperty(CIMProperty("x",
            Real32(1.5)));

        Array<Uint32> y;
        y.append(1);
        y.append(2);
        y.append(3);
        inst.addProperty(CIMProperty("y", y));
        Uint32 pos = inst.findProperty("v");

        if (pos != PEG_NOT_FOUND)
        {
            Uint32 v;
            CIMProperty prop =
            inst.getProperty(pos);
            prop.getValue().get(v);
        }
    }
    catch(...)
    {
    }
}
```

```
void CIMPLE_example()
{
    myClass* inst =
        myClass::create();
    inst->u.value = "hello";
    inst->v.value = 99;
    inst->w.value = true;
    inst->x.value = 1.5;
    inst->y.value.append(1);
    inst->y.value.append(2);
    inst->y.value.append(3);
    uint32 v = inst->v.value;
    destroy(inst);
}
```



Issue 2: CIM Operation Dichotomy

- **Instances** and **object paths** are parallel
- **BUT**
 - Implemented as separate CIM operations
 - Internally defined separately (CIMInstance vs. CIMObjectPath).
 - Parallel operation (enumerateInstance(), enumerateInstanceNames())
 - Create parallel work for provider developer
- **WHAT IF** objectPaths were treated as instances with just the key properties?
 - User defines instances, infrastructure manages corresponding paths



Example

- Short Example of creating paths, and instances in parallel
- NEED Example

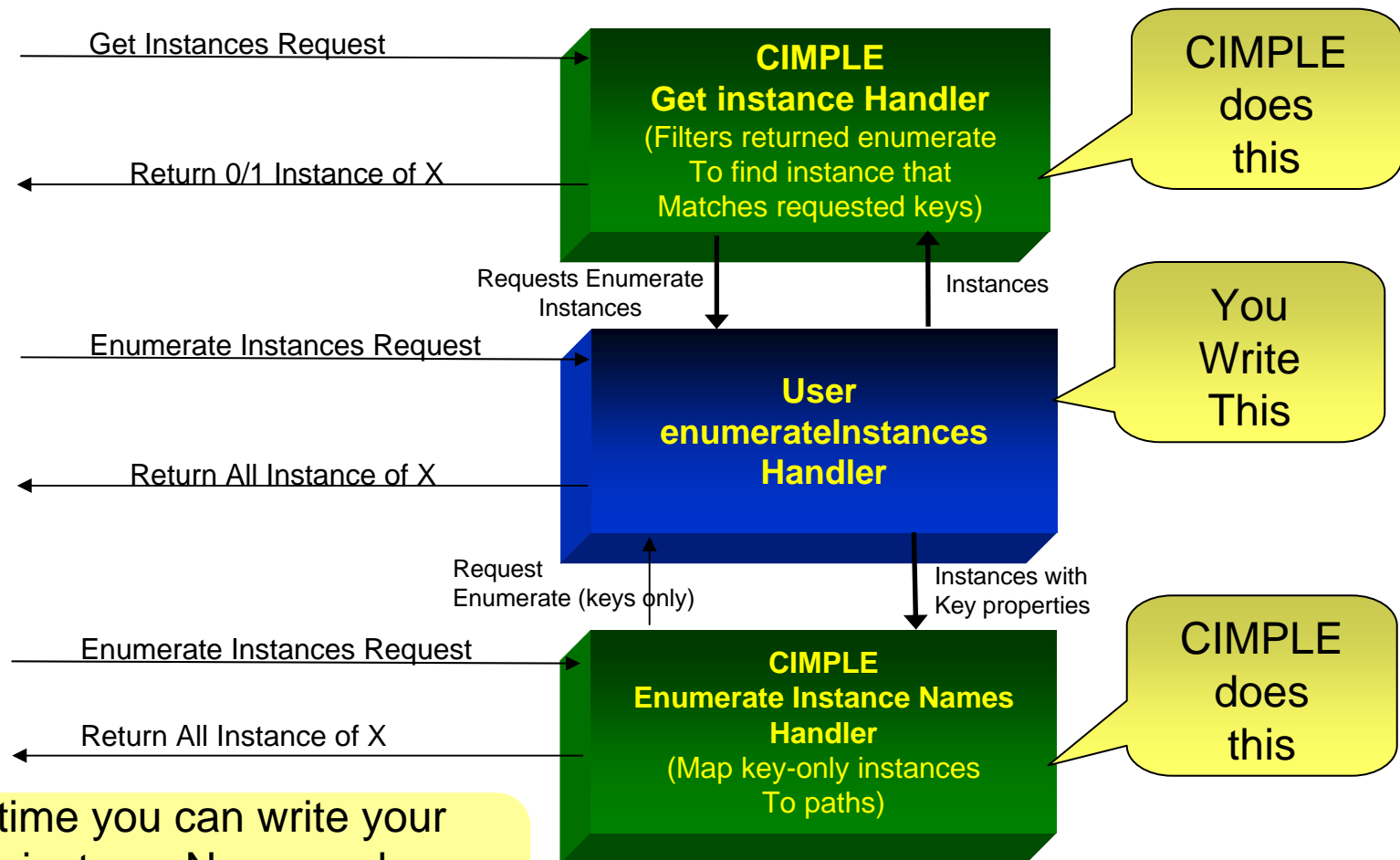
NOTE: Hiding this slide because not complete



Issue 3: Not all CIM operations should require programmer effort

- The core read operation is:
 - **EnumerateInstances**
- All other read operations could be derived from this
 - GetInstance
 - EnumerateInstanceNames
 - References
 - ReferenceNames
 - Associators
 - AssociatorNames

CIMPLE Reduces Instance Operation coding

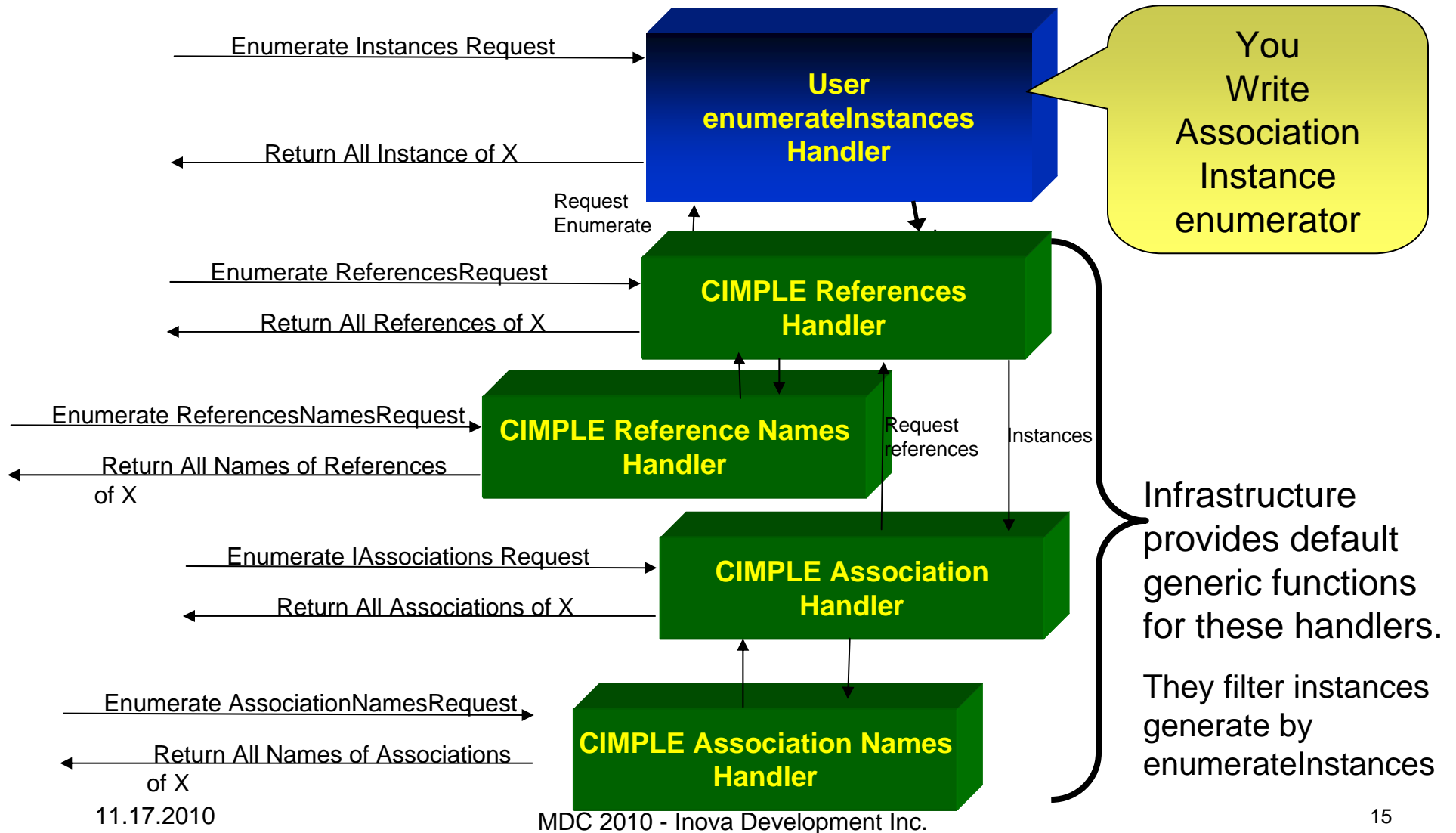


At any time you can write your Own instanceName and Get instance handlers
Functionality First, Performance later

Handling assoc/ref operations

- The assoc/ref operations are filters on `enumerateInstances`
 - Reference (filter instances that match target ref property, role and result class)
 - ReferenceNames (References reduced to just the `CIMObjectPath`)
 - Associators (filter instances that match target ref property, association class, result class, role, result role)
 - AssociatorNames (associators reduced to just the `CIMObjectPath`)
- **What if the developer only had to implement `enumerateInstances`?**
 - Write one CIM Operation all others default to implementations of filters provided by infrastructure
 - Later possibly implement the others as performance optimizations
 - Remember: most classes return very few objects.

Reducing Association Operations



Issue 4: Hand coding objects

- The provider writer normally manually builds dynamic instances (ex. In Pegasus the CIMInstance class).
 - Build from Class
 - getClass from repository
 - `CIMInstance x = myClass.buildInstance();`
 - Build by coding creation of dynamic objects
 - `CIMInstance myInstance("CIM_Junk");`
 - `addProperty(...);`
- Error prone activity

Concrete Classes

- What if the concrete class definitions were generated automatically from the mof?
- Advantages
 - Compile time error detection
 - Decreased code complexity
 - Assurance of code to MOF compatibility
 - Reduce code bloat
 - Improve program reliability

Concrete CIM Elements

- The **CIMPLE**, **BREVITY**, **konkret** interfaces employ **concrete CIM elements** whereas conventional interfaces use **abstract CIM elements**.

```
// Abstract CIM elements in Pegasus  
CIMInstance ci("Person");  
ci.addProperty(CIMProperty(  
    "Id", Uint32(1000)));  
ci.addProperty(CIMProperty(  
    "First", String("Homer")));  
ci.addProperty(CIMProperty(  
    "Last", String("Simpson")));
```

```
// Concrete CIM elements in CIMPLE:  
Person* inst = Person::create();  
inst->Id.set(1000);  
inst->First.set("Homer");  
inst->Last.set("Simpson");
```



Issue 5 – CIM Extrinsic Methods are hard to implement

- Provider writer must
 - Dynamically parse method names
 - Dynamically parse input parameters
 - Hand build methods to be executed and parse input parameters
 - Dynamically build response parameters and response return
- **What if?**
 - Infrastructure provided parse of method names and parameters
 - Created skeleton for the method to be invoked in native language (i.e C or C++ function)
 - Build output parameters and response return

Method Example

Pegasus InvokeMethod provider

```
void MethodProvider::invokeMethod(
    const OperationContext& context,
    const CIMObjectPath& objectReference,
    const CIMName& methodName,
    const Array<CIMParamValue>& inParameters,
    MethodResultResponseHandler& handler)
{
    CIMObjectPath localReference = CIMObjectPath(
        String(),
        CIMNamespaceName(),
        objectReference.getClassName(),
        objectReference.getKeyBindings());

    handler.processing();

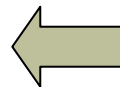
    if (objectReference.getClassName().equal("Foo"))
    {
        if (methodName.equal("Foo"))
        {
            if (inParameters.size() > 0)
                for (Uint32 i = 0; i < inParameters.size(); i++)
                {
                    if(inParameters[i].getName().equal("p1")
                    {
                        CIMValue paramVal = inParameters[i].getValue();
                        if (!paramVal.isNull())
                        {
                            ...
                        }
                    }

                    handler.deliverParamValue(CIMParamValue(
                        "p2",
                        CIMValue(String("blah"))));
                    handler.deliver(CIMValue(outString));
                }
            else
            {
                handler.deliver(0);
            }
        }
    }

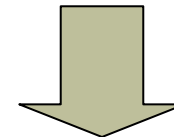
    handler.complete();
}
```

Example Class

```
class Foo
{
    [Key] uint32 key;
    uint32 foo(
        [In] string p1,
        [In(false), Out] string p2);
};
```



Dynamic Provider becomes C++ method



CIMPLE generated method

```
Invoke_Method_Status Foo_Provider::foo(
    const Foo* self,
    const Property<String>& p1,
    Property<String>& p2,
    Property<uint32>& return_value)
{
    .. Your code to set p2;
    return INVOKE_METHOD_OK;
}
```



Issue 6 - 90 – 10 Rule

- **90% of providers are very low volume**
- Only a small percentage of providers developed for a profile have high numbers of instances and require performance tuning
- At the same time, every provider takes significant work (or a complete infrastructure to simplify the task
 - With CIMPLe
 - Develop providers rapidly
 - Improve performance later
- Developers spend time on the important issues, not the repeated creation of many simple providers.

Goals

- **CIMPLE, BREVITY and KonkretCMPI** share the following goals:
 - Reduce code complexity.
 - Decrease development effort.
 - Reduce coding errors.
 - Improve reliability.
 - Promote interoperability.
 - Raise provider development from a detailed programming effort to a system development effort
 - Give the provider developer a common infrastructure so he can concentrate on his real problems:
 - Managing the issue of performance on those few providers that handle high volumes of instances
 - Managing the issue of the interface between the provider and the resource

Summary of Issues

- **Manual creation of providers**
 - Mof -> coder -> provider
- **Multiple similar CIM Operations**
 - Ex. Enumerate instance vs. enumerateinstancenames
- **Dynamic CIM object models**
 - Ex. Methods to create objects, etc.
- **Complex CIM operation set for provider**
 - Get, enumerate, references, associators ...
- **(10- 90 rule) Concentrate on the Important Provider issues**
 - Large volume
 - Difficult resource interfaces

The CIMPLE Vision

- **What**
 - Simplify provider writing
 - Reduce code complexity.
 - Decrease development effort.
 - Reduce coding errors.
 - Improve reliability.
 - Promote interoperability
- **HOW**
 - Automatic class/instance generation from MOF
 - Concrete class and instance generation
 - Automatic provider and provider module generation
 - Default CIM Operation generation
 - Remove basic dichotomies (ex. Instance and object paths)
 - Integrate registration of providers
 - Add missing convenience and support functions

CIMPLE?

- **CIMPLE** is an environment for developing providers with several key advantages.
 - They are easy to build.
 - They are small.
 - They are fast.
 - They are portable
- Key simplifications
 - Reduce the need to implement all of the provider operations.
 - Generate **real classes** in the target language **from MOF classes**.
 - Generate the provider skeleton **AND CIMOM INTERFACES** automatically.
 - Move error checking from runtime to compile time.
 - Separate the interface from the provider CIM Object manipulation



Licensing

- CIMPLe, BREVIty and KonkretCMPI released under **MIT open-source** license.
- Minimal restrictions on use.
 - Free to redistribute with license header retained in all sources.
 - No limitations on generated code
- No warranty license.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

NOTE: There is no licensing applied to code generated by these tools

Supported Platforms

OS	Processor	Compiler
Linux	IX86-32 bit	GNU C++
Linux	IX86-64 bit	GNU C++
Linux	PPC-32 bit	GNU C++
Linux	PPC-64 bit	GNU C++
Windows	IX86-32 bit	MSVC++
Windows	IX86-64 bit	MSVC++
Darwin	IX86-32 bit	GNU C++
Solaris	SPARC	GNU C++
VxWorks 6.X	XScale	GNU C++
VxWorks 5.X	Pentium	GNU C++

Note: KonkretCMPI not tested in windows and uses linux installation utilities

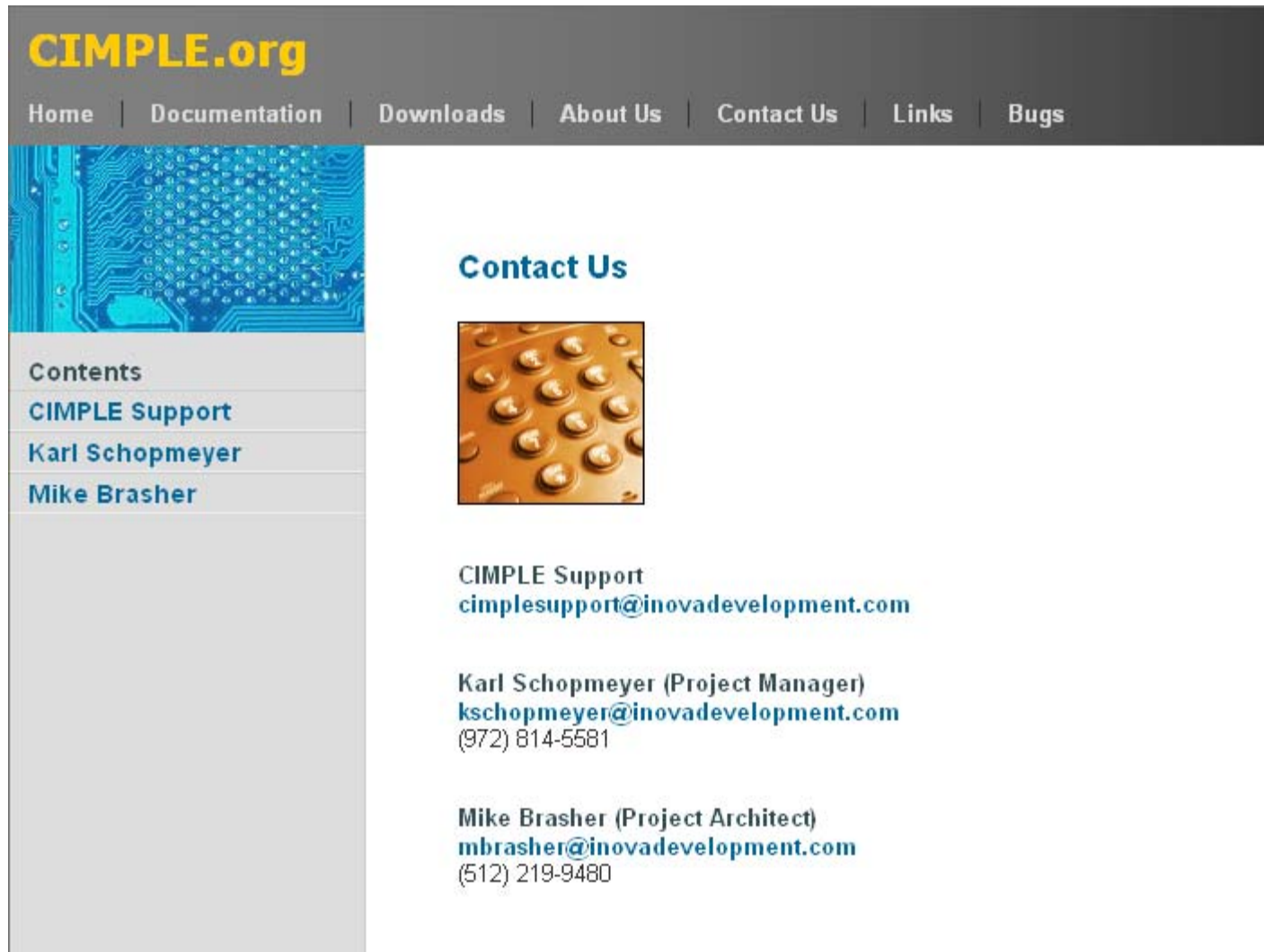


What's new

- CIMPLe (CIMPLe 2.0 2008)
 - Interface to WMI CIM Object Manager
 - Expanded multithread support
 - Expanded error responses
 - Bug fixes
- CIMPLe 2.0.2 – 2.0.16 (2008 – now)
 - Bug fixes, 64 bit support, etc.
- CIMPLe 2.1 (end of year)
 - Memory Cache
 - Persistent Instance Repository
 - Expanded Threading Support
 - Instance Repository
 - Bug fixes
 - Dynamic control of configuration from within provider
 - Improved diagnostic/debugging support



<http://simplewbem.org>


A screenshot of the CIMPLE.org website. The header is dark grey with the site name 'CIMPLE.org' in yellow. A navigation menu includes 'Home', 'Documentation', 'Downloads', 'About Us', 'Contact Us', 'Links', and 'Bugs'. A sidebar on the left contains a blue circuit board image and a list of links: 'Contents', 'CIMPLE Support', 'Karl Schopmeyer', and 'Mike Brasher'. The main content area is titled 'Contact Us' and features an image of orange pill capsules. Below the image, contact information is provided for 'CIMPLE Support', 'Karl Schopmeyer (Project Manager)', and 'Mike Brasher (Project Architect)', including their email addresses and phone numbers.

CIMPLE.org

Home | Documentation | Downloads | About Us | Contact Us | Links | Bugs

Contents
CIMPLE Support
Karl Schopmeyer
Mike Brasher

Contact Us



CIMPLE Support
cimplesupport@inovadevelopment.com

Karl Schopmeyer (Project Manager)
kschopmeyer@inovadevelopment.com
(972) 814-5581

Mike Brasher (Project Architect)
mbrasher@inovadevelopment.com
(512) 219-9480



Support

- **Free Support**
- Answer questions via e-mail sent to cimplesupport@inovadevelopment.com.
- Accept patches subject to review and approval.
- Accept bug reports (open Bugzilla database).
- Fix bugs based on our own priorities and objectives.
- **Paid support** available through Inova Development.
- Paid support includes:
 - Quick turnaround on support questions.
 - Quick resolution to bugs.
 - Implementation of custom enhancements.
 - Special releases to address client needs.
 - “Emergency” support for tight delivery schedules.
 - Adding extensions to the environment



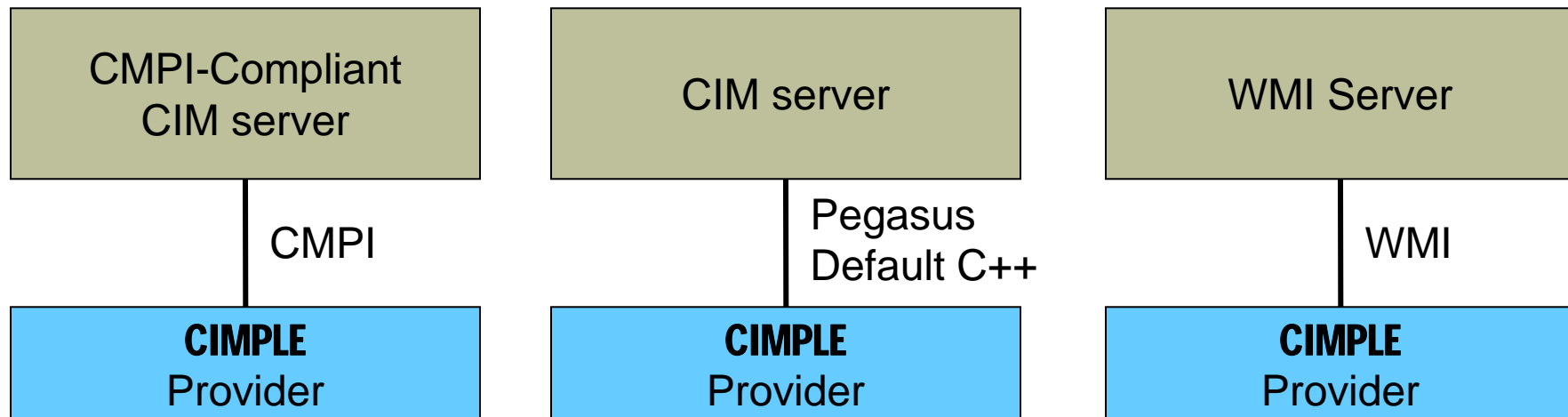
Obtaining CIMPLE, BREVITY, and BREVITY

- Freely available:
 - Download source distributions
 - Since these tools are for developers we felt that source distribution should not be an issue
- See
 - www.inovadevelopment.com
 - Simplewbem.org
 - Konkretcmpi.org

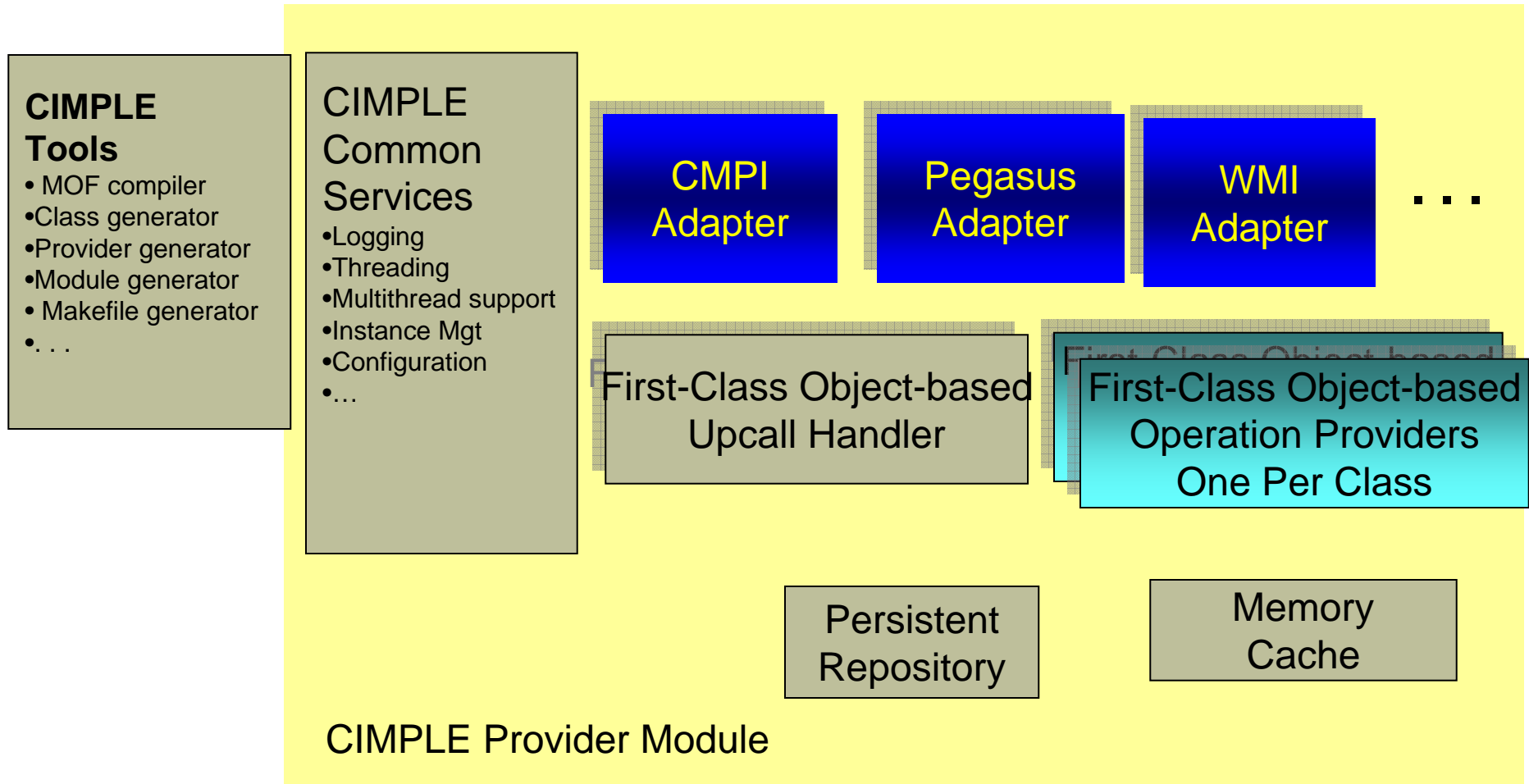
CIMPLE

What is CIMPLe?

- **CIMPLe** is an open-source tool for building CIM providers for multiple provider interfaces (and multiple CIM servers).
- **CIMPLe** providers can be reconfigured to support multiple provider interfaces (without source code changes).



CIMPLE Provider Architecture



CIMPLE CMPI Providers

- CMPI Providers created with CIMPLE are true CMPI providers.
 - They implement the CMPI provider interface.
 - They define the proper CMPI entry points.
 - They can be loaded by CMPI-capable CIM servers.
 - Integrated registration for Pegasus



CIMPLE Pegasus C++ Providers

- Pegasus Default C++ Providers created with CIMPLE are true Pegasus C++ interface providers.
 - Implement Pegasus C++ interfaces
 - Implement Pegasus C++ entry point
 - Registered and loaded by Pegasus

A blue 3D rectangular box with a yellow border, containing the text 'Pegasus Adapter' in yellow. The box is positioned in the bottom right area of the slide.

Pegasus
Adapter

CIMPLE WMI Providers

- Pegasus WMI created with CIMPLE are true WMI interface providers.
 - Implement Microsoft WMI interfaces
 - Implement WMI entry point
- Register as com provider with Microsoft com tools
- The generated module.cpp is no different except for generate entry point
- CIMPLE genmod generates
 - The Microsoft .def file
 - The register.mof file (register with mofcomp)
 - Makefile slightly different to account for linking Microsoft libraries





CIMPLE Provider Types

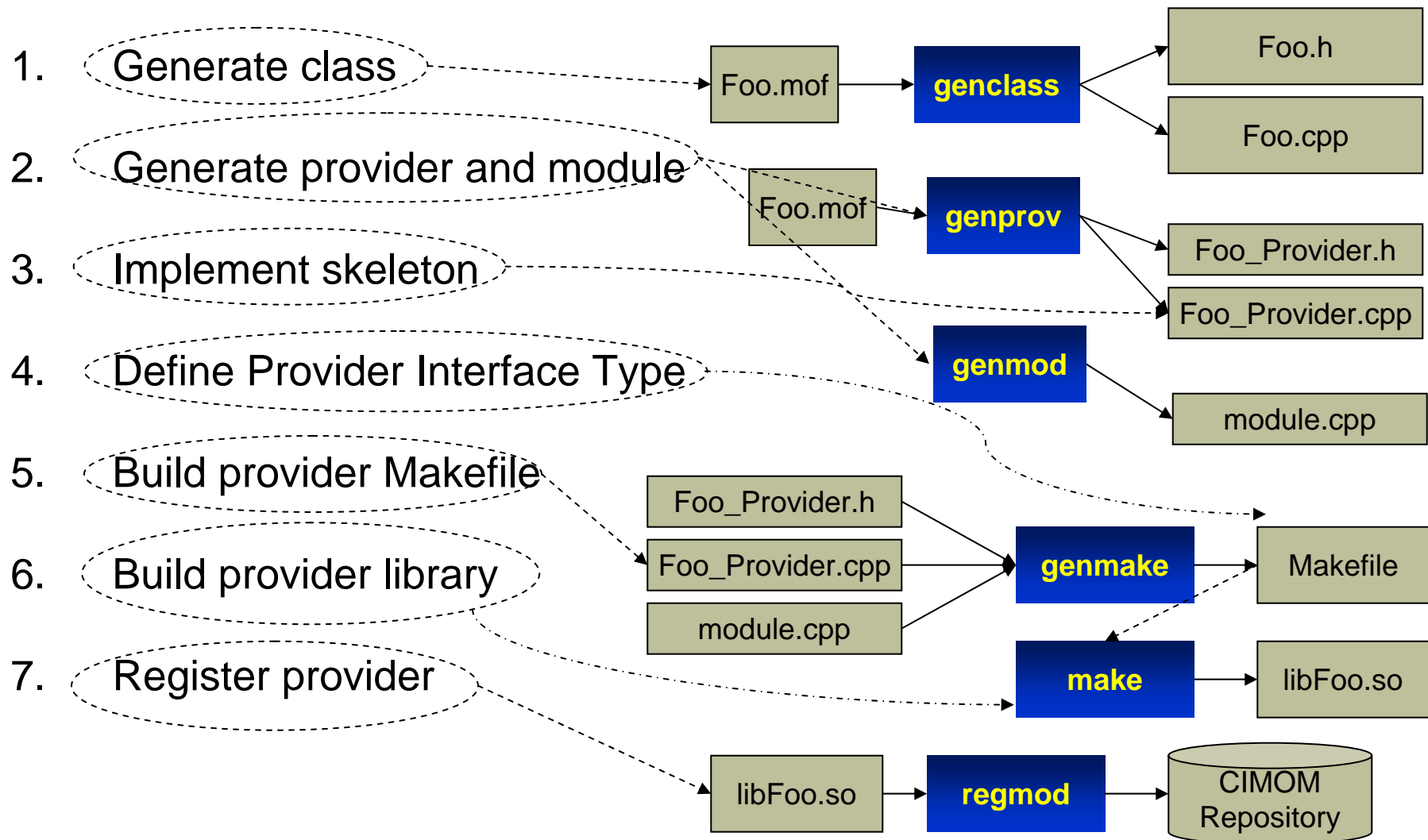
- **Instance Providers**
- **Association Providers**
- **Method Providers**
 - Note: This is no separate method provider type in CIMPLE. Any CIMPLE instance provider generates methods defined in the MOF
- **Indication Providers**



Major Features

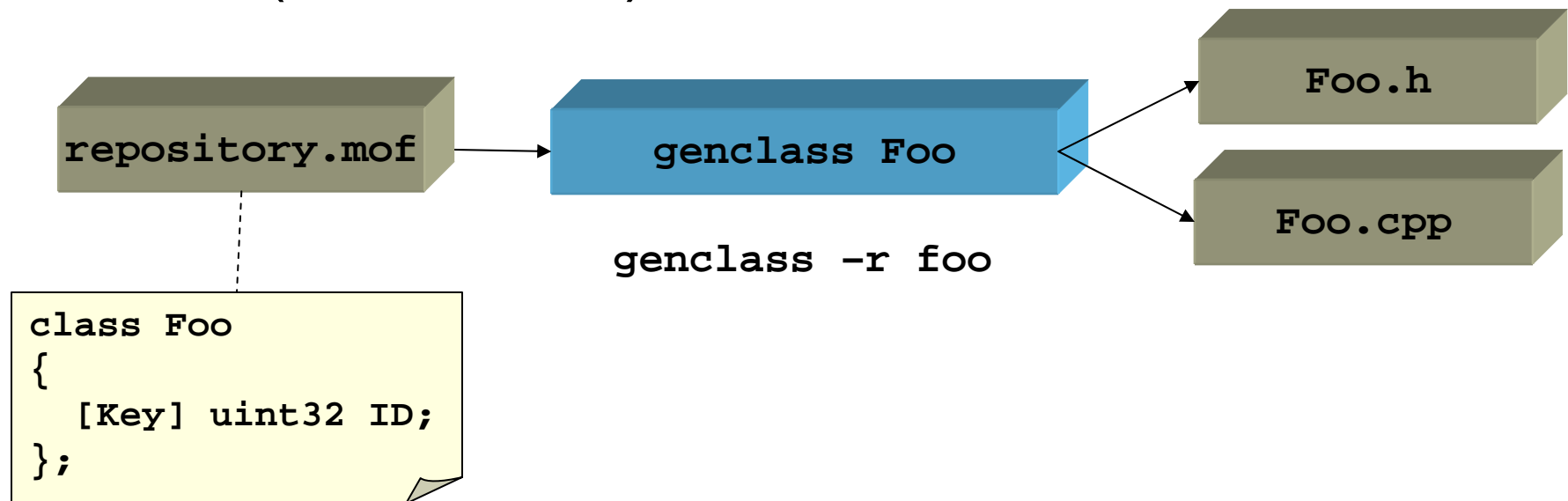
- C++ Class generation (from MOF).
- Provider skeleton generation.
- Extrinsic method stub generation.
- Provider Module generation.
- Provider module Makefile generation.
- Automated provider registration.
- Elimination of object paths.
- Operation automation.
- Module packaging
- Multiple CIMOM/Provider Interfaces

CIMPLE Provider Development Steps



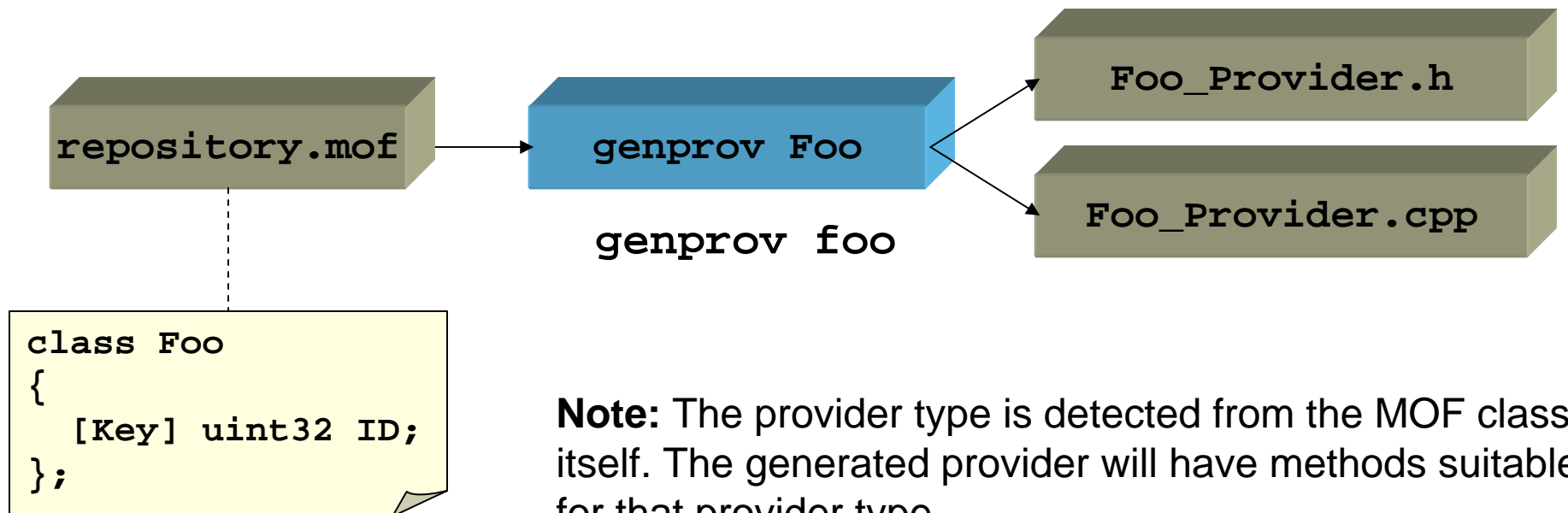
Class Generation

- CIMPLE generates C++ files defining the class (meta-data) for each class

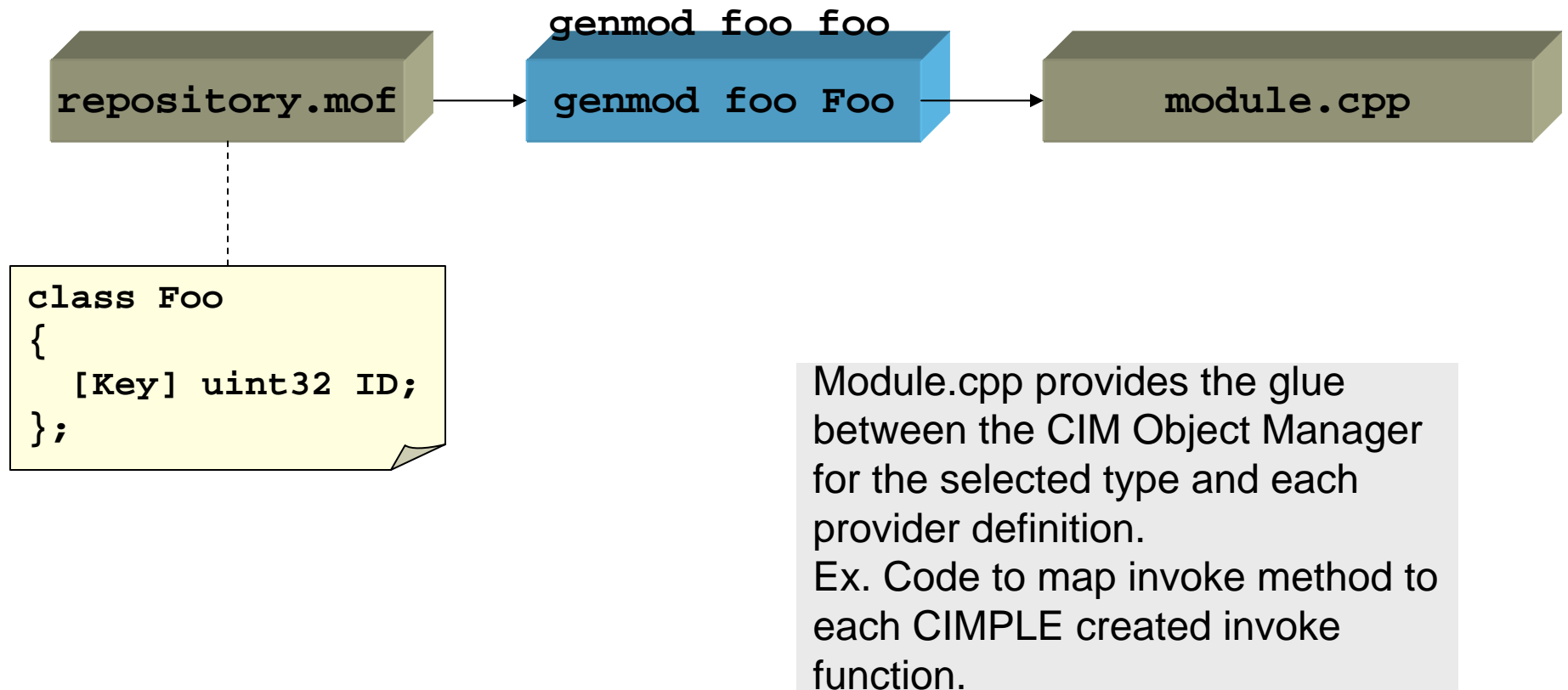


Provider Generation

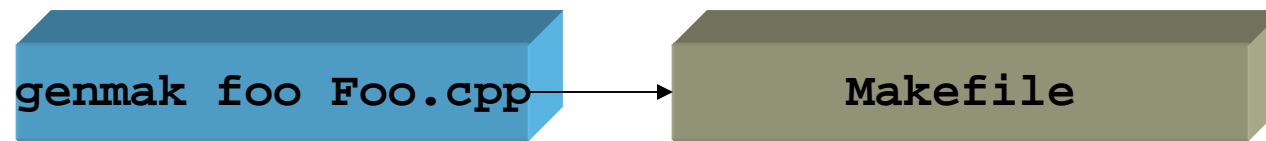
- **CIMPLE** generates the provider skeletons automatically for provider operations and extrinsic methods.



Module Generation

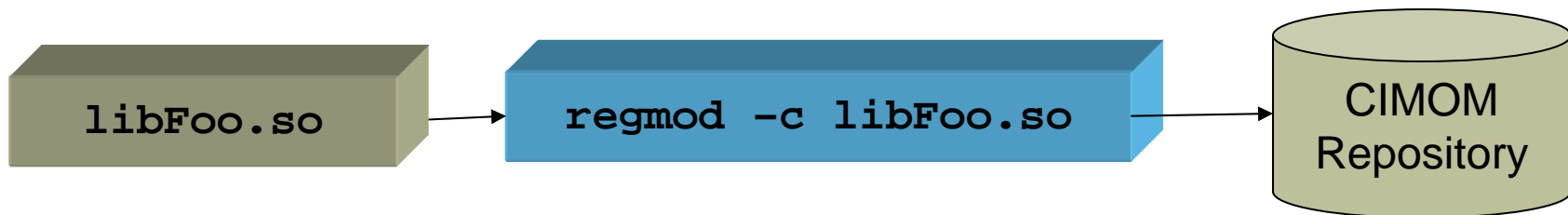


Provider Module Makefile Generation



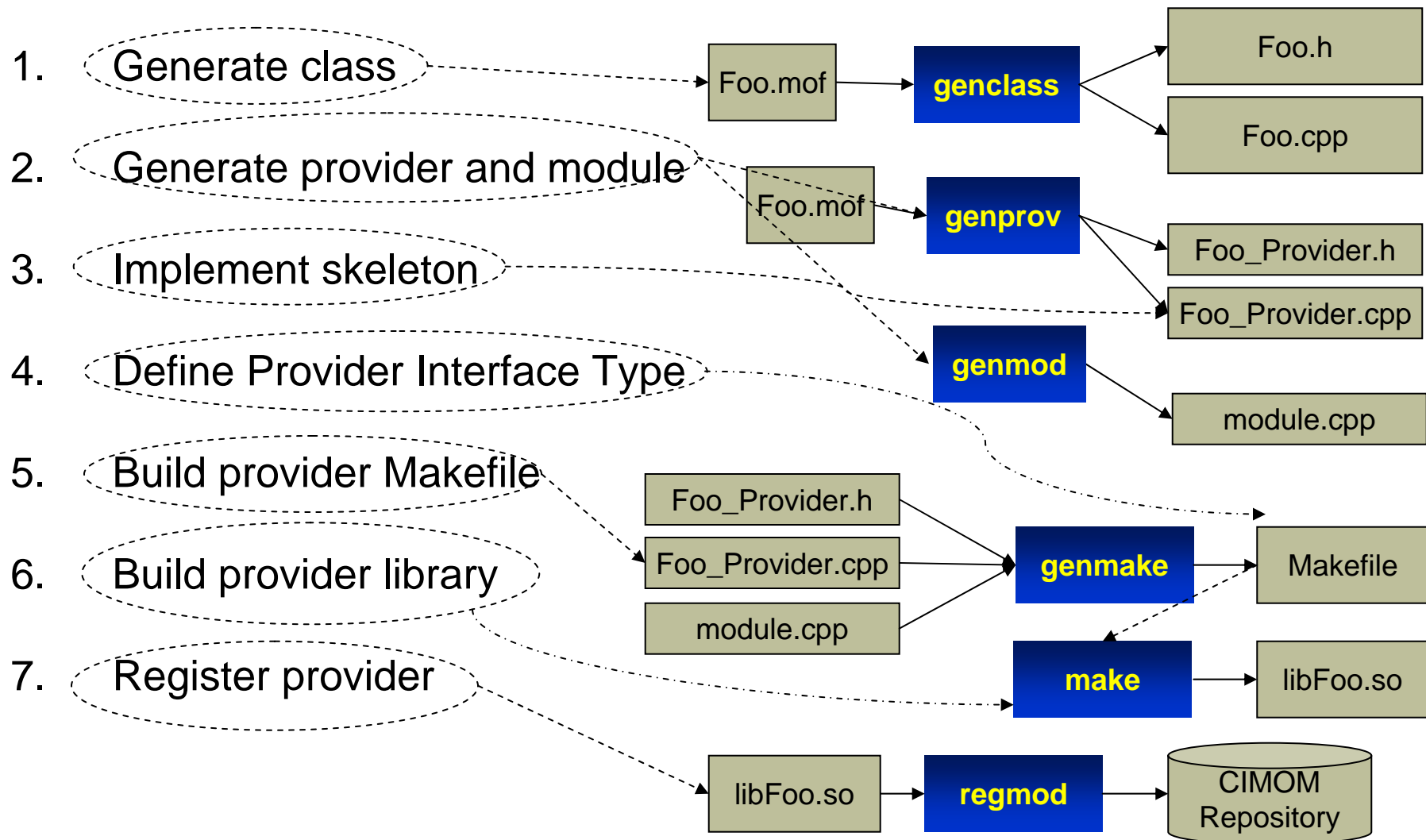
Provider Module Registration

- Regmod tool automates provider registration and class creation
- CIMPLE integrates registration into the library file.



1. Regmod only works with Pegasus today.

CIMPLE Provider Development Steps

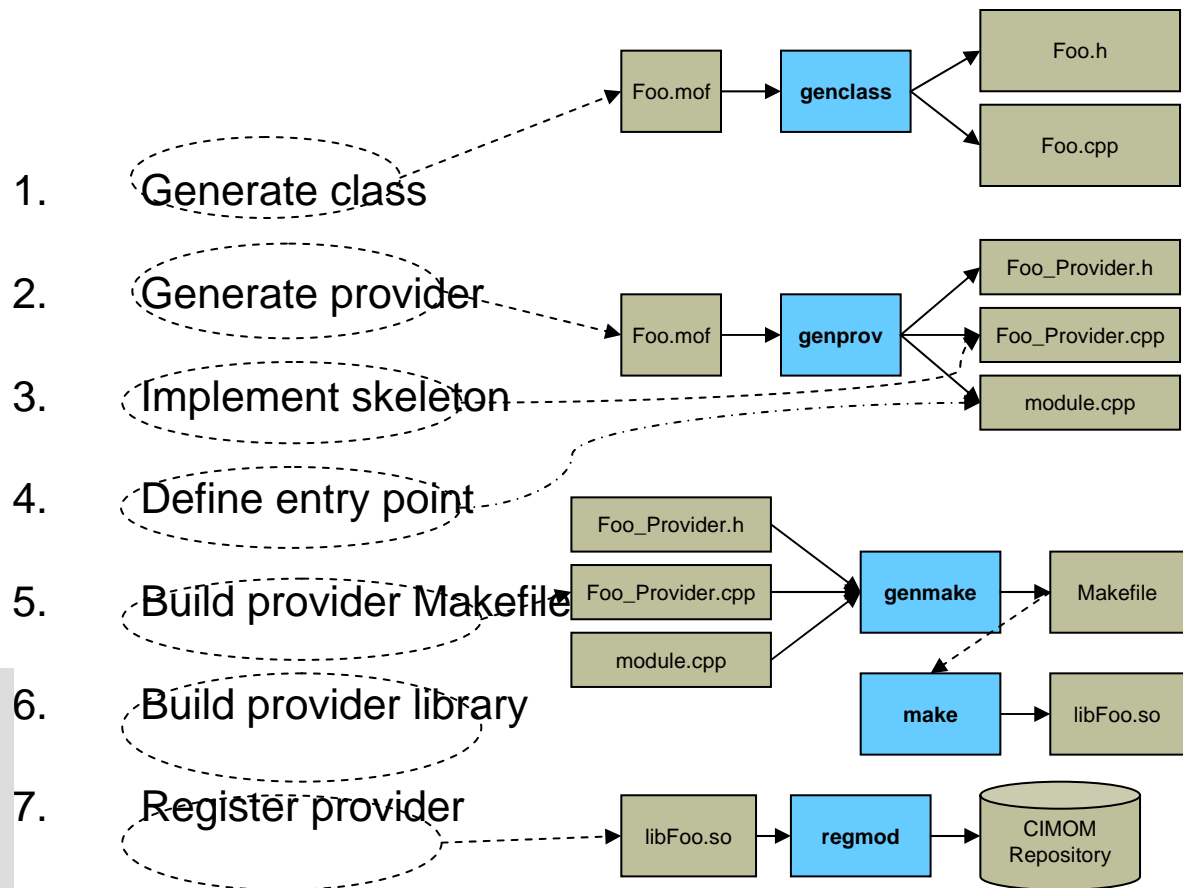


Genproj Tool

**genproj <provider>
<class list>**

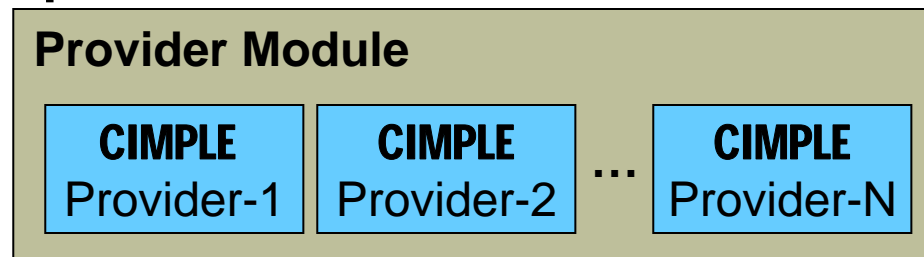
Genproj replaces serial use of:

- genclass
- genprov
- genmod
- genmake



CIMPLE Provider Modules

- CIMPLE allows several providers to be packages together in a **provider module**.
- A provider services the operations for a single class.
- Modules contained in a single shared library.
- `regmod` utility discovers and registers all providers in a provider module.



Object Path Elimination

- In **CIMPLE**, object paths are represented with ordinary instances. Only the key fields are used. This eliminates the confusing dichotomy between an object path and an instance. For example.

```
class MyClass
{
  [Key] uint32 Key1;
  [Key] string Key2;
  uint32 Prop3;
  string Prop4;
  boolean Prop5;
};
```

```
// MyClass.Key1=99,Key2="Hello"
MyClass* keys = MyClass::create();
keys->Key1.value = 99;
keys->Key2.value = "Hello";
```



Operation Elimination and Automation

- **CIMPLE** eliminates the need for several operations.
 - Get-instance-names
 - Enumerate-instance-names
 - Associator-names
 - Reference-names
 - Invoke-method
- **CIMPLE** automates the following operations via enumerate-instances, if the provider developer opts not to implement them.
 - Get-instance (uses enumerate-instances).
 - Associator-names (uses enumerate-instances).
 - References (uses enumeration-instances).

Example

- Outline
 - Generate provider skeletons.
 - Implement enumerate-instances operation.
 - Implement associators operation.
 - Implement extrinsic method.
 - Install and register provider.
 - Verify provider using client tools.

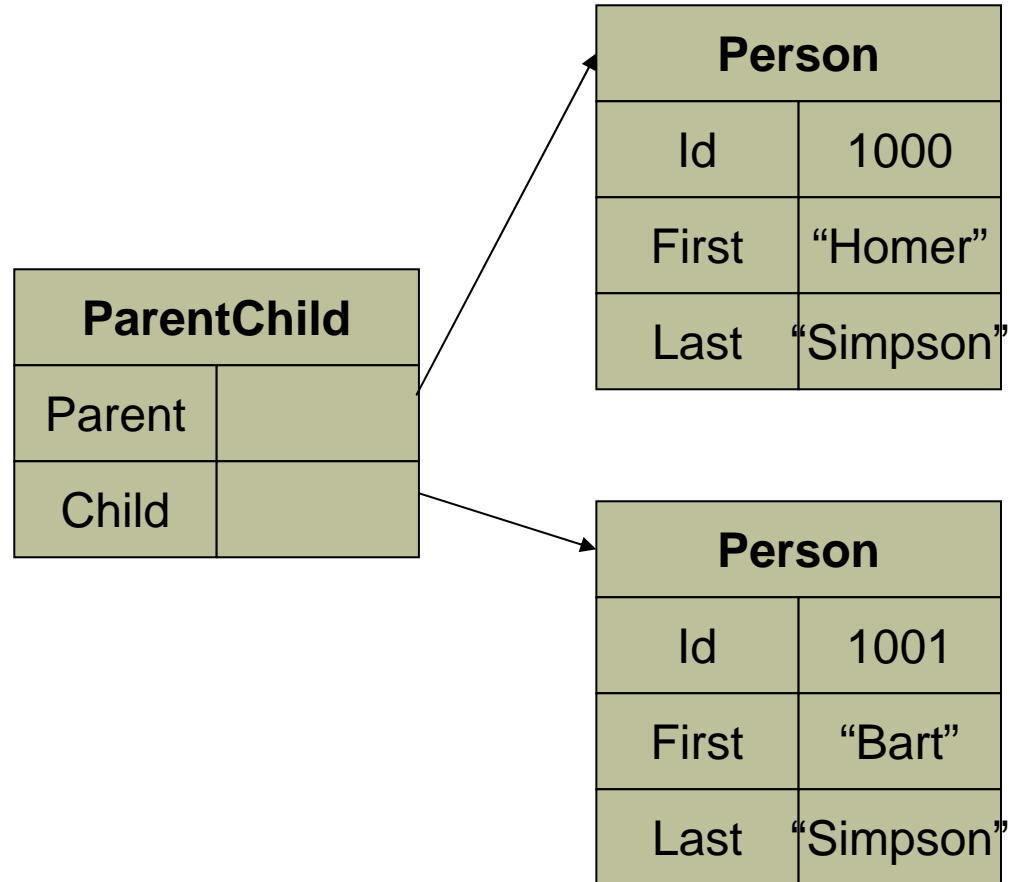
Example Classes & Instances

```

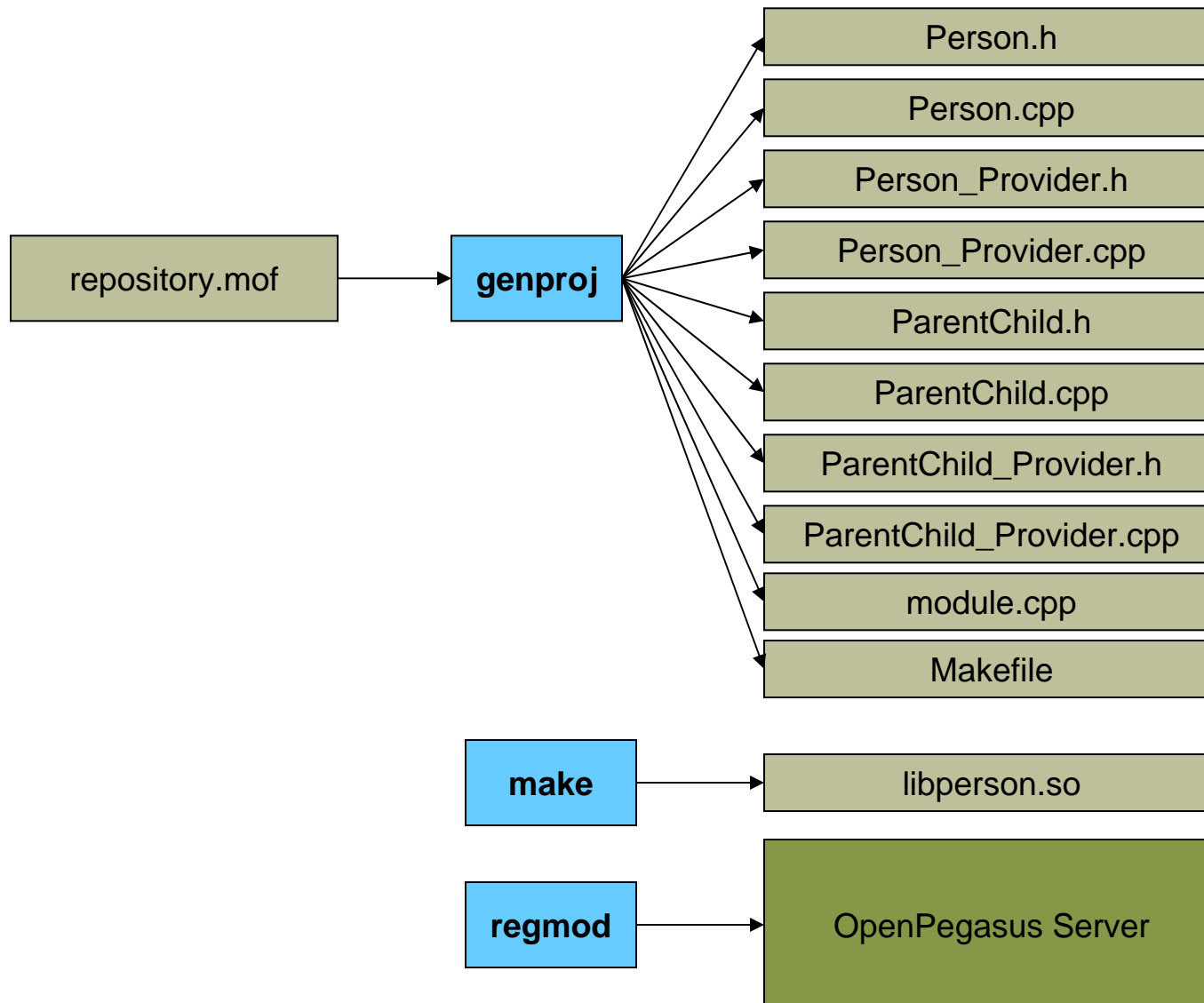
class Person
{
  [Key] uint32 Id;
  string First;
  string Last;

  uint32 GetProperties(
    [In(false), Out] uint32 Id,
    [In(false), Out] string First,
    [In(false), Out] string Last);
};

[Association]
class ParentChild
{
  [Key] Person REF Parent;
  [Key] Person REF Child;
};
  
```



Example Workflow



Instance Provider

Person_Provider.cpp

```
Enum_Instances_Status Person_Provider::enum_instances(  
    const Person* model,  
    Enum_Instances_Handler<Person>* handler)  
{  
    Person* p1 = Person::create();  
    p1->Id.value = 1000;  
    p1->First.value = "Homer";  
    p1->Last.value = "Simpson";  
    handler->handle(p1);  
  
    Person* p2 = Person::create();  
    p2->Number.value = 1001;  
    p2->First.value = "Bart";  
    p2->Last.value = "Simpson";  
    handler->handle(p2);  
  
    return ENUM_INSTANCES_OK;  
}
```

Your code

Association Provider

ParentChild_Provider.cpp

Your
Code

```
Enum_Instances_Status ParentChild_Provider::enum_instances(  
    const ParentChild* model,  
    Enum_Instances_Handler<ParentChild>* handler)  
{  
    Person* homer = Person::create();  
    homer->Id.set(1000);  
  
    Person* bart = Person::create();  
    bart->Id.set(1001);  
  
    ParentChild* assoc = ParentChild::create();  
    assoc->Parent = homer;  
    assoc->Child = bart;  
  
    handler->handle(assoc);  
  
    return ENUM_INSTANCES_OK;  
}
```

Extrinsic Method

Person_Provider.cpp

```
Invoke_Method_Status Person_Provider::GetProperties(  
    const Person* self,  
    Property<String>& First,  
    Property<String>& Last,  
    Property<uint32>& return_value)  
{  
    return_value.null = false;  
    return_value.set(1);  
    if (self->Id.value == 1000)  
    {  
        First.set("Homer");  
        Last.set("Simpson");  
        return_value.set(0);  
    }  
    if (self->Id.value == 1001)  
    {  
        First.set("Bart");  
        Last.set("Simpson");  
        return_value.set(0);  
    }  
    return INVOKE_METHOD_OK;  
}
```

Your code

Generator installs
INVOKE_METHOD_UN SUPPORT

Indication Providers

- CIMPLe Implements
 - enableIndications()
 - disableIndications()
 - deliverIndication()
 - Indications become CIMPLe instances
 - Hesitated to implement much of the filter analysis, etc. tooling because of inconsistent model of usage with different CIM object managers.



CIM Object Manager Upcalls

- First class object based upcalls to server to get information from other providers
 - Enumerate instances
 - Get instances
- Returns are CIMPLE objects
- Note. This code is already driven through iterators so ready for future pull operations

Memory Cache

- Limited version (Instance_Map) today.
Expanded version planned for 2.1
- Cache CIMPLE instances in memory store
 - Smaller than most other storage forms
- Semantics for
 - Insert, find, enumerate, delete, modify
- Multiple patterns for usage
 1. Resource -> cache. Cache -> CIM Operation
 2. Timed cache – CIMPLE automatically gets from cache if it exists (future)
 3. Cache some properties, actively acquire other. Merge to satisfy request.

Memory Cache Example

```
Enum_Instances_Status Person_Provider::enum_instances(const Person* model,
    Enum_Instances_Handler<Person>* handler)
{
    return _map.enum_instances(model, handler);
}

Create_Instance_Status Person_Provider::create_instance(Person* instance)
{
    if (instance->ssn.null == true)
    {
        return CREATE_INSTANCE_FAILED;
    }
    return _map.create_instance(instance);
}

Delete_Instance_Status Person_Provider::delete_instance(const Person* instance)
{
    return _map.delete_instance(instance);
}

Modify_Instance_Status Person_Provider::modify_instance(const Person* model,
    const Person* instance)
{
    return _map.modify_instance(model, instance);
}
```

Persistent Store

- Interface consistent with cache
 - Find, insert, delete, modify, enumerate
- Persist in disk file with checkpoint/ recovery
- File location determined by .cimplerc config file
- File name (file & index) by class



Other Support Services

- Threading
- Thread communication
 - Mutexes
 - Condition Variables
 - Condition Queues
 - Task Scheduler
 - Atomic variables
- Logger
- Lists and stacks
- Arrays
- Get information from context
- Runtime configuration file



Short Term Plans

- Version 2.1 Release (End this year)
 - Instance Repository - Store and retrieve instances on persistent storage
 - Expand memory Cache
 - Add dynamic control of some provider functionality
 - Ex. Control log output and levels

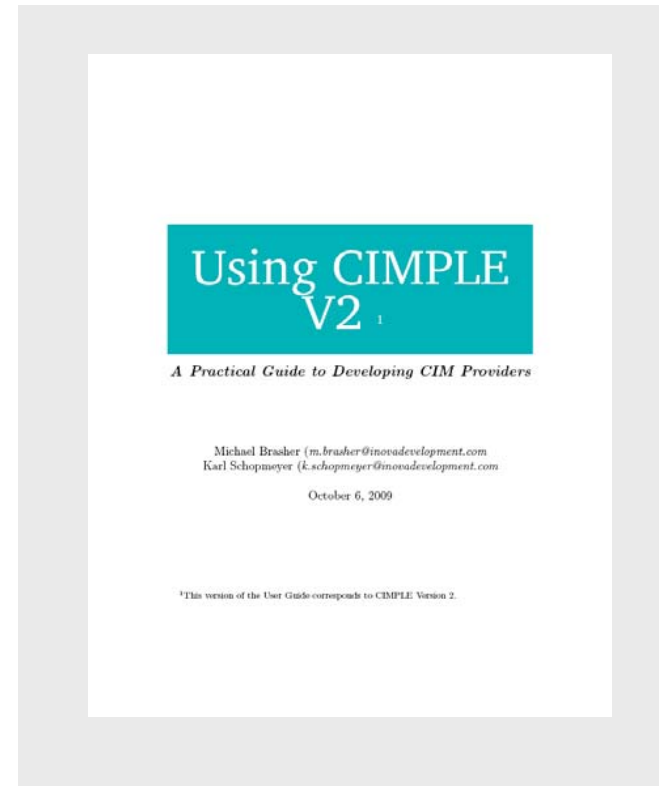


Long Term Plans

- Move more to profile generation
- Work with CMPI spec upgrade when this become available
- Expand to other patterns of provider function
 - Ex. Merge properties from cache, persistent store, source into an instance.
- Improve debugging capabilities (ex.)
 - Resource usage
 - Provider state
 - Dynamic control parameters to provider
- Expand Return capabilities (CIM_Error, Std Msg Support

CIMPLE Documentation

- Using CIMPLE Book
 - Overview
 - Examples
 - API definitions
 - Maintained current with each release
 - Part of download
 - Or directly on web site.
 - It's a really good read 😊





TBD

- Indication Provider
- Etc.
- Other APIs in the kit
- What CIMPLe Includes
- CIMPLe Documentation
- Major goal – Portability of real providers between environments.
- Handling qualifiers, etc.
- Automate functionality of modify instance
- An instance size



Retargeting a CIMPLE Provider

- **Retargeting normally involves NO code changes**
 - Modify Target variable in Makefile
 - Rebuild module.cpp and link.
 - Module.cpp provides the point of entry
- **Provider interface choice defined by variable on the provider compile and link**
 - CIMPLE_PEGASUS_MODULE, CIMPLE_CMPI_MODULE, CIMPLE_WMI_MODULE
 - Set this (ex in Makefile CIMPLE_CMPI_MODULE=1)
- **Or with genmake to retarget the provider for CMPI:**
\$ genmak -C

...

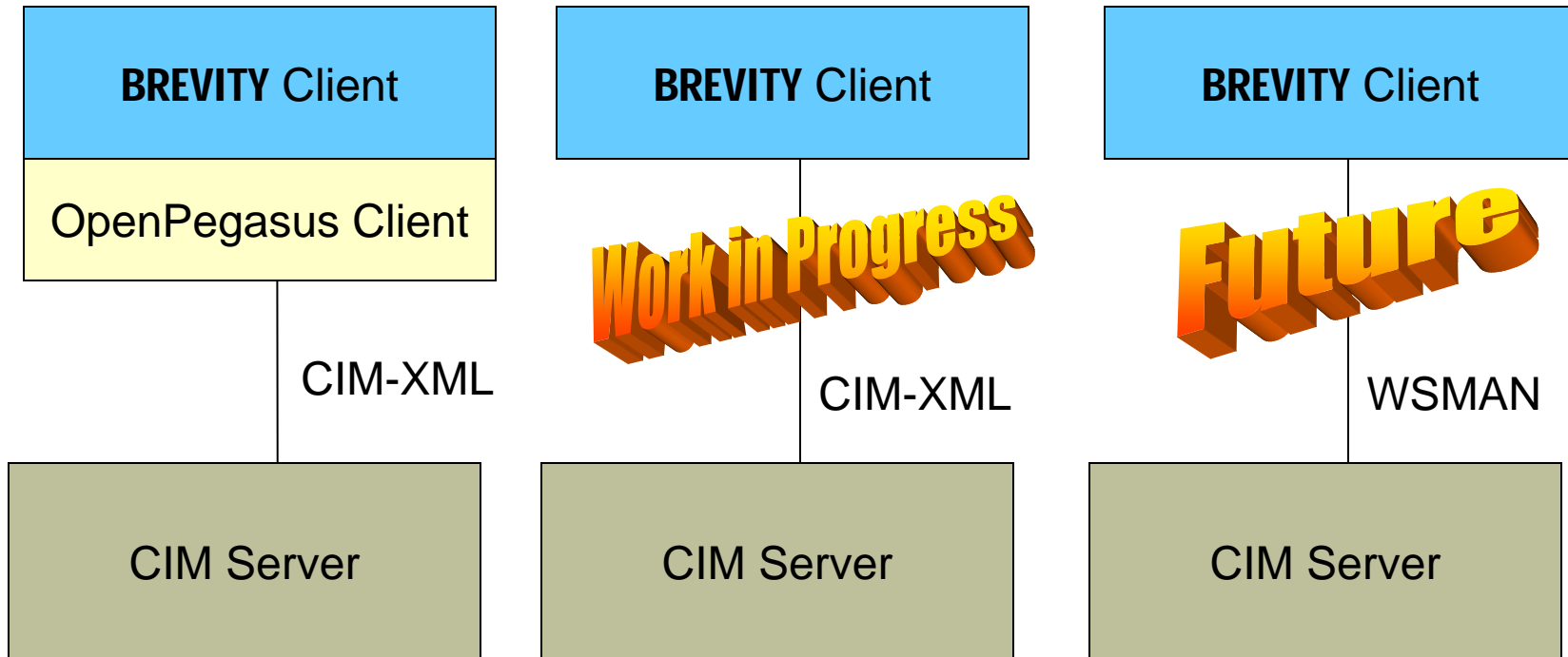


CIMPLE Advantages Summary

- **Support for multiple provider interfaces**
 - CMPI
 - Pegasus C++ provider interface
 - WMI
 - ...
- **Create Providers for multiple CIM Servers**
 - Pegasus, SFCB, WMI
- **Reduce development effort and cost**
 - Our problem is the infrastructure, yours is the resource and model information
- **Reduce code complexity**
 - CIMPLE provides infrastructure, object infrastructure, default operations, and many common functions (ex. Threads, logging)
- **Fewer development errors**
- **Smaller and faster providers (embedded systems)**

What is BREVITY?

- **BREVITY** is an open-source environment for developing C++ CIM clients.

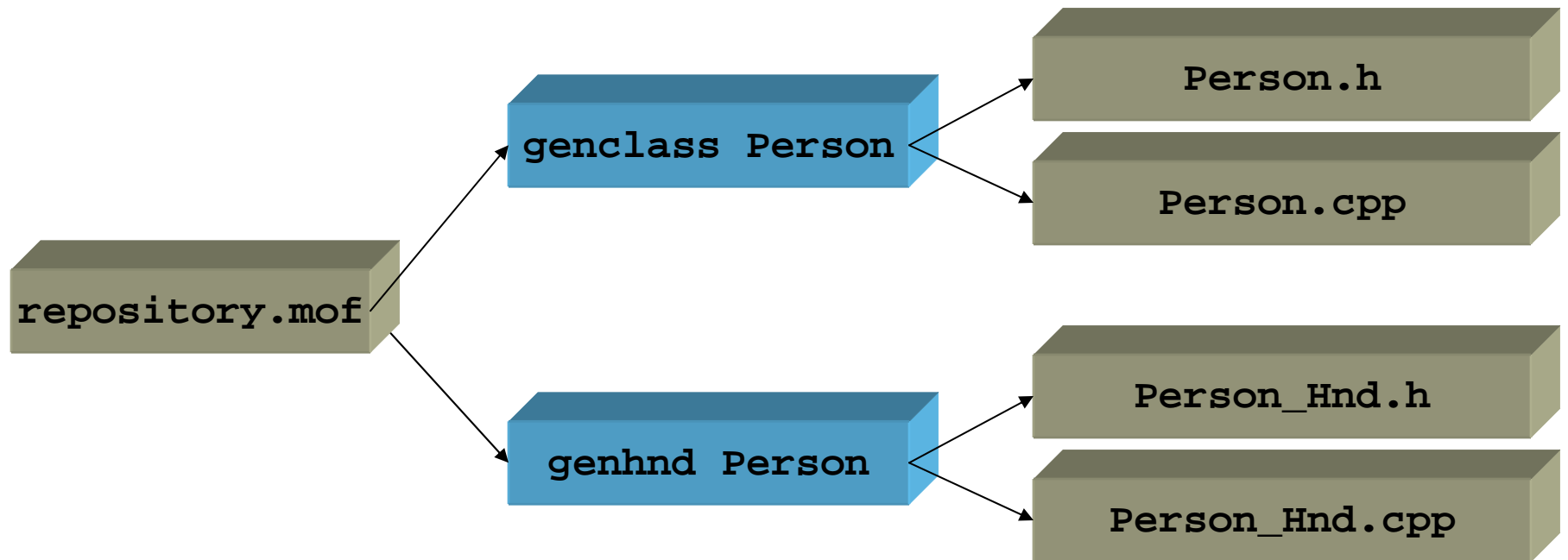




BREVITY Advantages

- Less development effort and cost.
- Reduced code complexity
- Fewer development errors.
- Support for multiple CIM servers.
- Generates concrete classes.
- Generates extrinsic method stubs.

Concrete Class Generation





Working with Generated Class Handles

```
// Create and initialize an instance of Person.  
Person_Hnd h;  
h.Id_value(1000);  
h.First_value("Homer");  
h.Last_value("Simpson");  
h.print();
```



Working with Generated Class References

```
// Create Person.Id=1000:  
Person_Ref r;  
r.Id_value(1000);
```

Enumerating Instances

```
Client c;  
c.connect();  
  
Instance_Enum e = c.enum_instances("root/cimv2", Person_Ref());  
  
while (e.more())  
{  
    Person_Hnd person(e.next());  
    person.print();  
}
```



Invoking an Extrinsic Method with BREVITY

```
Person_Ref ref;  
ref.Id_value(1000);  
  
Arg<String> first;  
Arg<String> last;  
Arg<uint32> result = ref.GetProperties(c, "root/cimv2", first, last);  
  
printf("result=%u:%u\n", result.value(), result.null());  
printf("first=%s:%u\n", first.value().c_str(), first.null());  
printf("last=%s:%u\n", last.value().c_str(), last.null());
```

Invoking an Extrinsic Method With Pegasus

```
// Build reference for "Person.id=1000"

Array<CIMKeyBinding> keyBindings;
keyBindings.append(CIMKeyBinding("Id", "1000", CIMKeyBinding::STRING));
CIMObjectPath ref(String(), CIMNamespaceName(), "Person", keyBindings);

// Invoke the method:

Array<CIMParamValue> in;
Array<CIMParamValue> out;
CIMValue resultValue = c.invokeMethod(
    "root/cimv2", ref, "GetProperties", in, out);

// Print result:

if (resultValue.getType() != CIMTYPE_UINT32)
{
    // Handle error!
}

Uint32 result;
resultValue.get(result);
printf("result: %u:%u\n", result, resultValue.isNull());

// Check number of output arguments:

if (out.size() != 2)
{
    // Handle error!
}

// Print the output arguments:
```

```
for (Uint32 i = 0; i < out.size(); i++)
{
    const CIMParamValue& param = out[i];

    if (String::equalNoCase(param.getParameterName(), "First"))
    {
        CIMValue firstValue = param.getValue();

        if (firstValue.getType() != CIMTYPE_STRING)
        {
            // Handle error!
        }

        String first;
        firstValue.get(first);
        printf("first: %s:%u\n", (const char*)first.getCString(),
            firstValue.isNull());
    }
    else if (String::equalNoCase(param.getParameterName(), "Last"))
    {
        CIMValue lastValue = param.getValue();

        CIMValue lastValue = param.getValue();

        if (lastValue.getType() != CIMTYPE_STRING)
        {
            // Handle error!
        }

        String last;
        lastValue.get(last);
        printf("last: %s:%u\n", (const char*)last.getCString(),
            lastValue.isNull());
    }
    else
    {
        // Handle error (unknown output parameter)!
    }
}
```



KonkretCMPI



What is KonkretCMPI

- **KonkretCMPI** 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

Pitfalls with CMPI interface

- The dynamic nature of CMPI interfaces
- CMPI code complexity
- Instance and instance path dichotomy
- Method handling complexity
- Support for common functions
- Too many operations

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

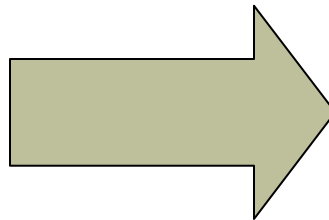
- Instances can be converted to CMPI Instances
 - CMPIInstance* instance;
 - CMPIStatus* status;
 -
 - instance = WidgetToInstance(&w, &status);

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
- Minimum implementation is only `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)



Where to use CIMPLE or Konkret

- **CIMPLE**

- C++
- Supports multiple provider interfaces (CMPI, C++ pegasus, WMI)
- Common provider code between multiple interfaces
 - Change with link
- Supports Windows platforms
- Includes extensive support for threading, etc. to keep common code.

- **GOAL**

- General provider development
- Support multiple provider interfaces with same provider

- **Konkret**

- C language
- CMPI interface
- *nix supported platforms
- Providers will run anywhere a cmapi provider would run
- No additional runtime libraries required for deployment

- **GOAL**

- Embedded CMPI provider environments
- Can be used with existing CMPI providers (in the same provider)
- Allow concrete object, skeleton creation, etc. without using c++ and within cmapi provider philosophy



Future Directions

- **CIMPLE**
 - Active development today
 - More common features for the provider environment
 - Availability of memory cache, instance persistent repository, configuration control (V 2.1)
 - Handle CIM_Error objects with errors
 - Reduce provider size further
 - Size measuring tools(instance size, etc.)
 - Improve tracing and debugging tool set
 - Long term
 - Grow from provider development to profile development
 - Interface to scripting languages as a resource.
 - Query Providers
 - Match Specification Changes (ex. Pull operations)
 - WE ARE ACTIVELY LOOKING FOR FUTURE IDEAS FROM OUR USERS
- **Brevity**
 - Now integrated with CIMPLE
- **KonkretCMPI**
 - Parallel growth to CIMPLE but probably more limited since it live within CMPI framework.

Questions & Discussion



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