CIMPLE
A CIM Provider Engine

Mike Brasher, Karl Schopmeyer

Inova Development

January 16, 2006

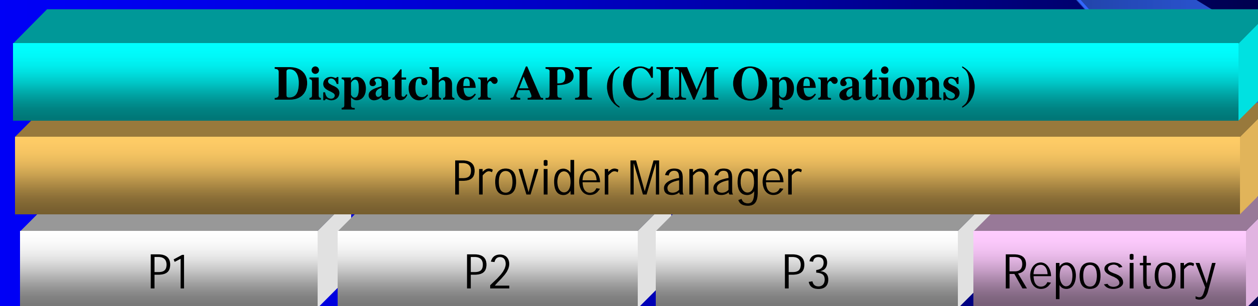
Version 1.0

Introduction

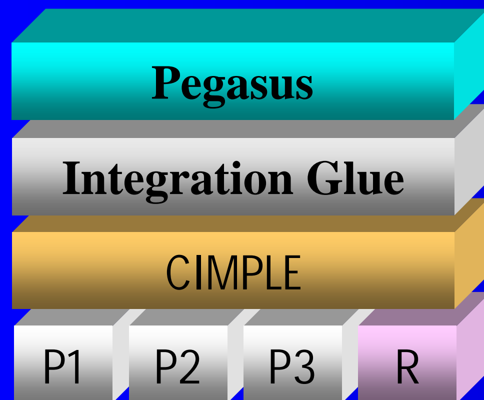
What is CIMPLE?

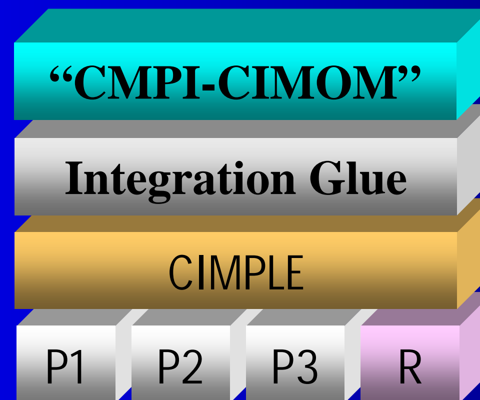
- CIMPLE is a miniature CIM provider engine.
  - Embeddable.
  - Light-weight.

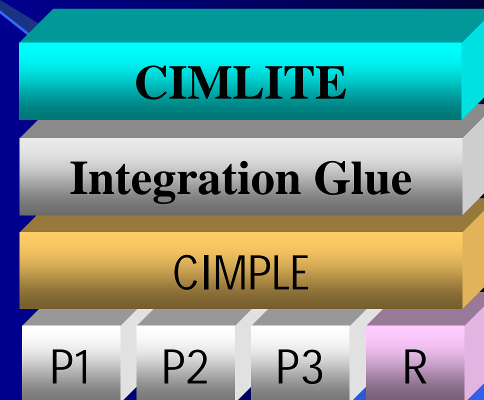
CIMPLE Architecture

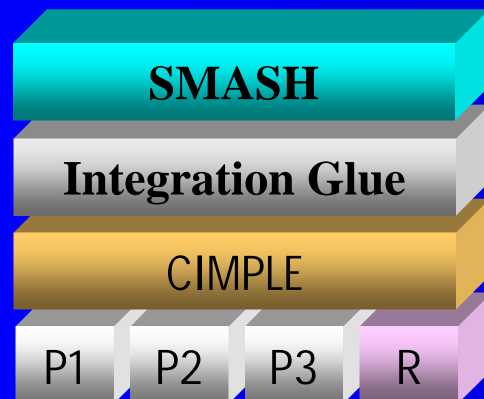


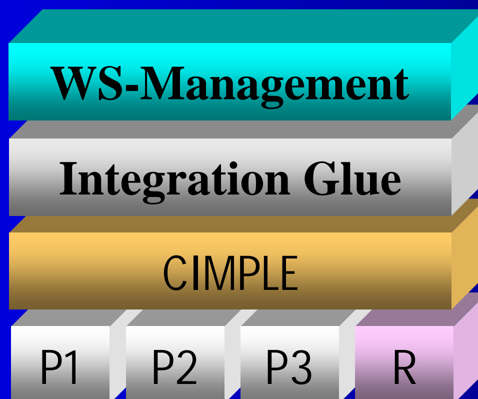
CIMPLE Applications

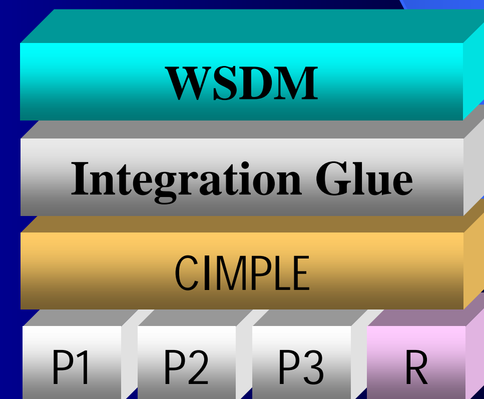








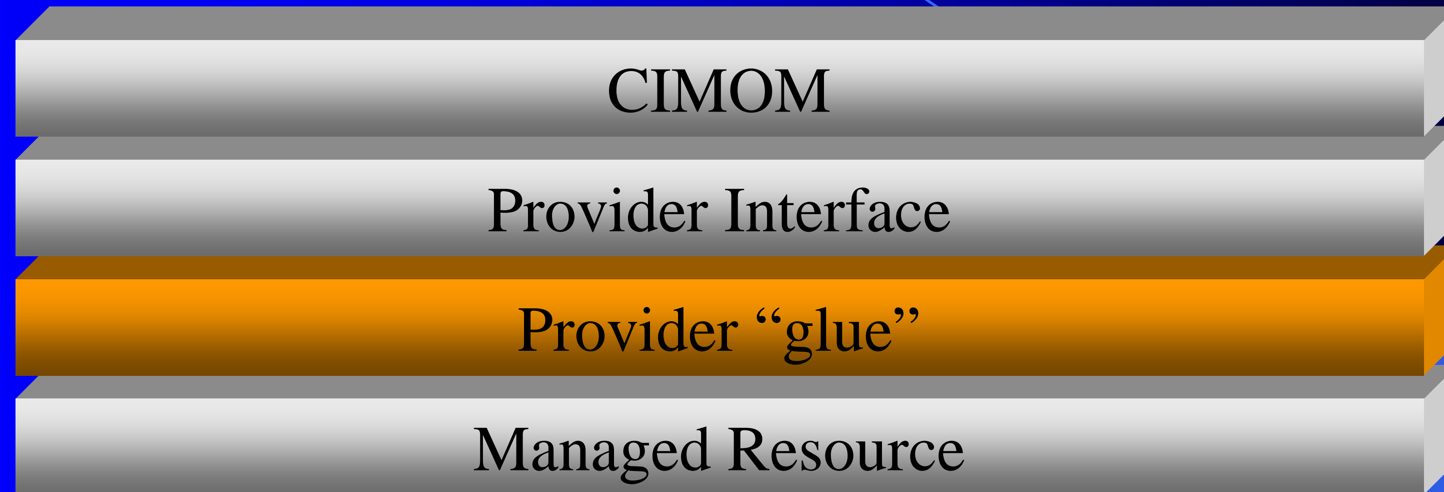




Advantages

- CIMPLE is used to develop providers with four key advantages.
  - Easy to build
  - Small
  - Fast
  - More reliable

Conventional Provider



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

Key simplifications

- Eliminates the need to implement several of the provider operations.
- Generates real classes in the target language from MOF classes.
- Generates the provider skeleton and CIM interface automatically.

Provider Operations

- Get-instance
- Get-instance-names*
- Enum-instance
- Enum-instance-names*
- Get-property*
- Set-property*
- Create-instance
- Modify-instance
- Delete-instance
- Associators*
- Associator-names*
- References*
- Reference-names*
- Method-stubs*
- Deliver indication

*CIMPLE provides these operations

Class generation

CIMPLE generates classes in the target language from MOF classes.

```
class Fan
{
    [key] string DeviceID;
    uint64 Speed;
    uint64 DesiredSpeed;

    uint64 SetSpeed(
      [in] uint64 speed);
};
```

```
class Fan
{
    Value<string> DeviceID;
    Value<uint64> Speed;
    Value<uint64> DesiredSpeed;
    CIMPLE_CLASS(Fan);
};
```

Provider skeleton generation

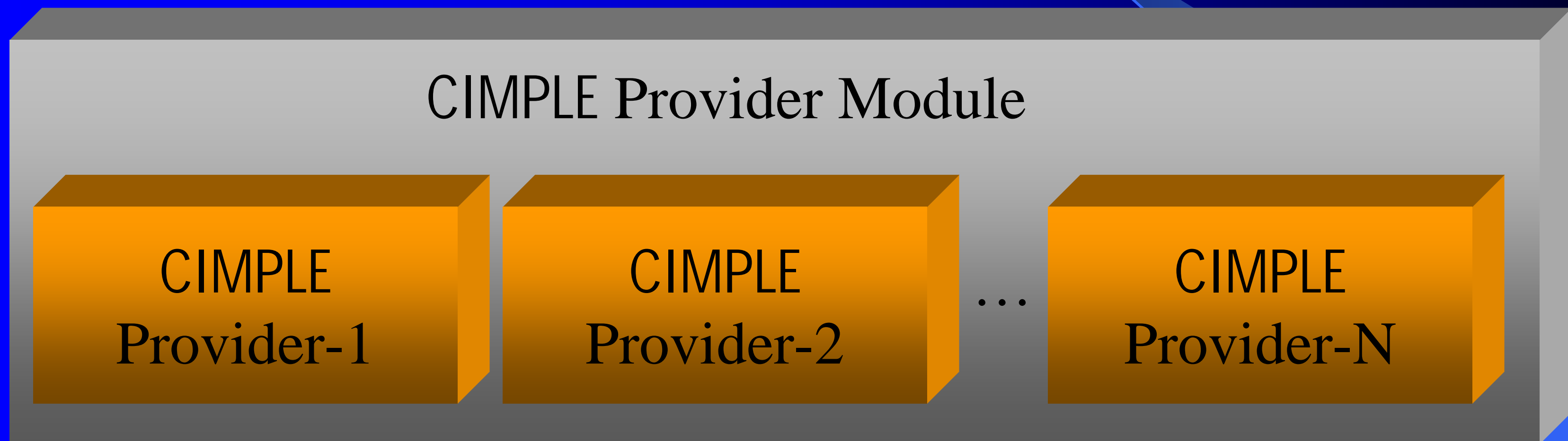
- Generates skeletons for instance operations.

```
Get_Instance_Status Fan_Provider::get_instance(
    const Fan* model,
    Fan* instance)
{
    return GET_INSTANCE_UNSUPPORTED;
}
```

- Generates method signatures and method dispatchers.

```
Invoke_Method_Status Fan_Provider::SetSpeed(
    const Fan* inst,
    const Value<uint64>& DesiredSpeed,
    Value<uint32>& return_value)
{
    return INVOKE_METHOD_UNSUPPORTED;
}
```

CIMPLE Module Architecture



Motivation

The Problem

- Conventional providers are:
  - **hard** 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).
- Management of the provider infrastructure dominates the provider development effort.

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");
    }
}
```

What if Fan were a real class?

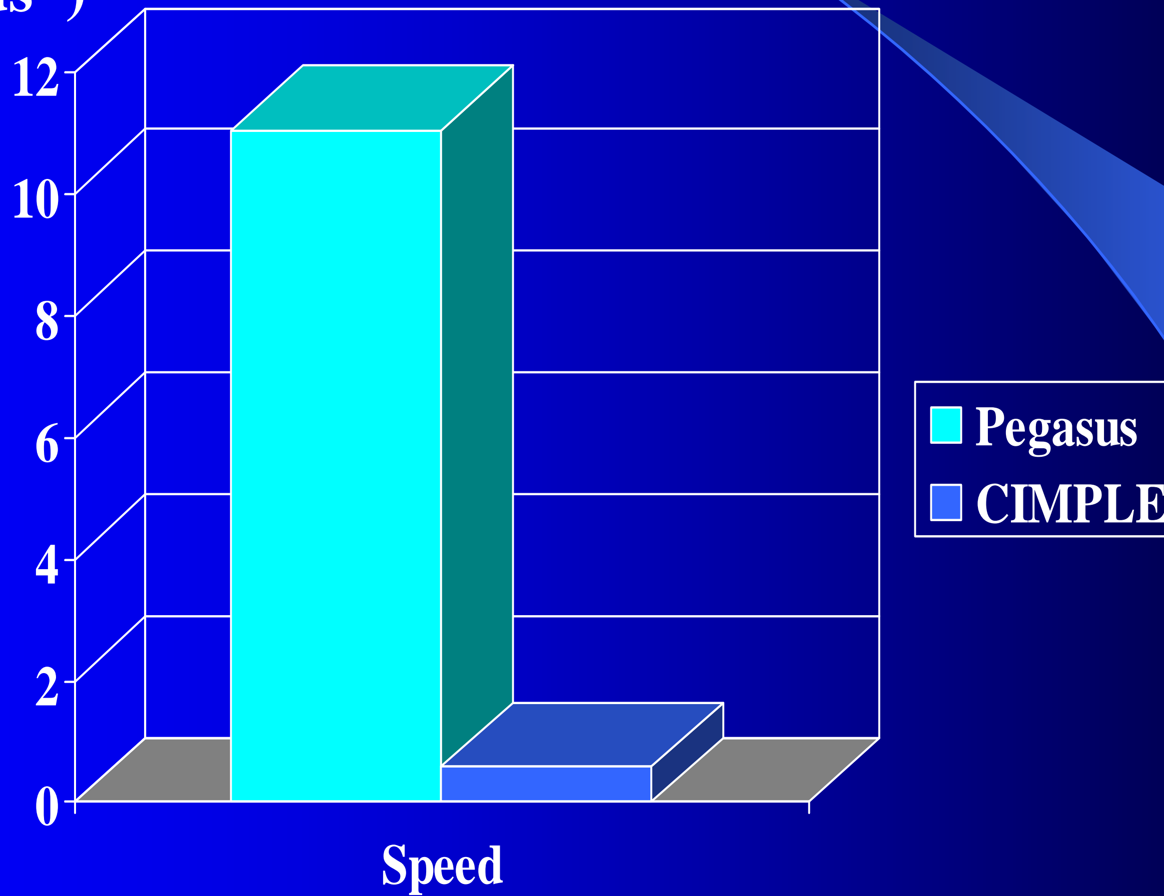
```
inline Uint32 GetSpeed(const Fan& fan)
{
    return fan.Speed.value;
}

// Or just ‘fan.Speed.value’
```

Pegasus/CIMPLE

```
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(...)
  {
  }
}
```

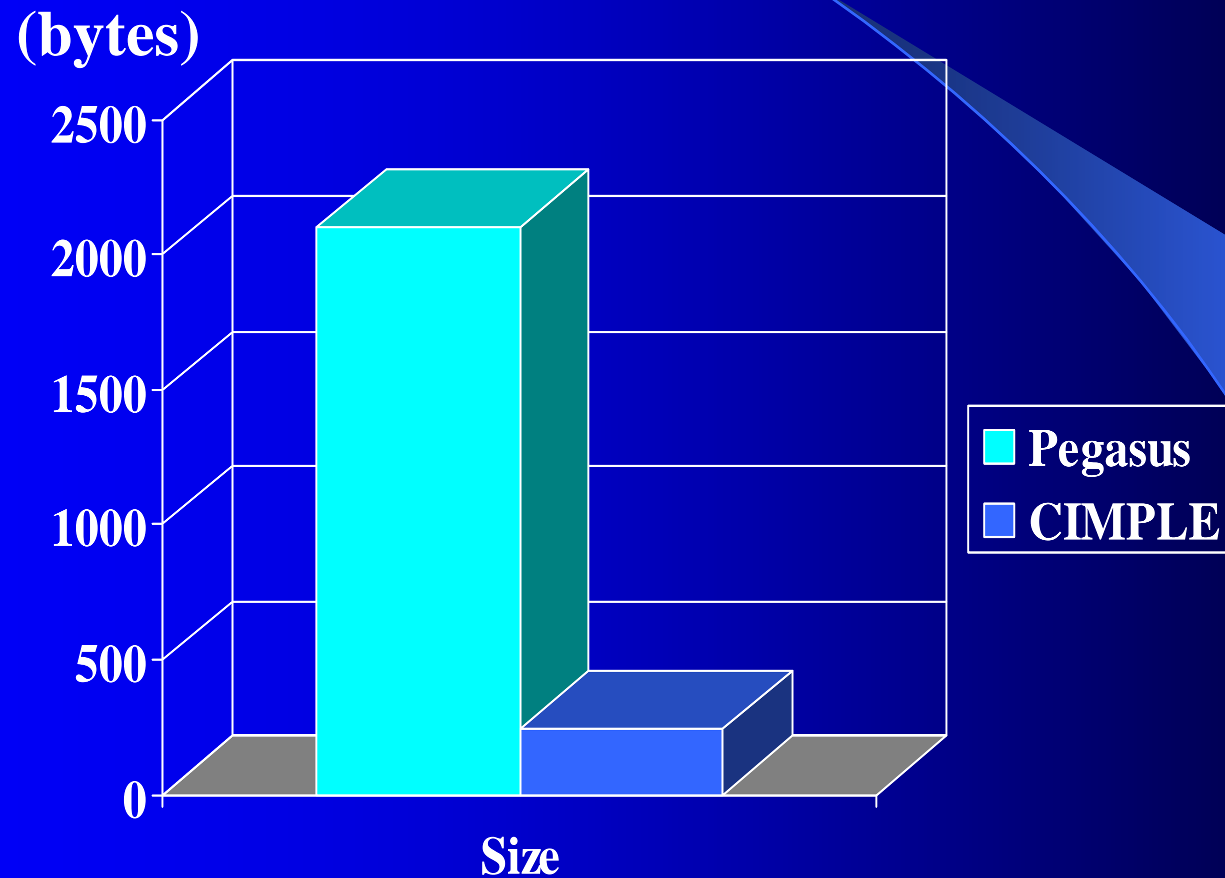
```
void CIMPLE_example()
{
  TheClass* inst = TheClass::create();
  inst->u.value = "hello";
  inst->v.value = 99;
  inst->w.value = true;
  inst->x.value = 1.5;
  inst->y.value.append(1);
  inst->y.value.append(2);
  inst->y.value.append(3);
  uint32 v = inst->v.value;
  destroy(inst);
}
```

Speed Comparison



* One million iterations (18X)

Size Comparison



* (9X)

Advantages

CIMPLE advantages

- **Easier** – eliminates busy work, allowing one to concentrate on the core problem.
- **Smaller** – reduces object size and memory usage.
- **Faster** – improves performance.
- **Reliable** – catches at compile time rather than run-time.
- **Interoperable** – could potentially run under any CIMOM.

Easier

- Development easier with ***first class objects***.
- Fewer instance provider operations to implement.
- No need to implement association operations.
- Method signatures automatically generated.
- Tools automate complex activities:
  - **genclass** – generates C++ classes from MOF.
  - **genprov** – generates provider skeletons.
  - **regmod** – registers provider with Pegasus.
  - **testmod** – pre-loads and tests providers.

Smaller

- CIMPLE providers have smaller **object size**.
- CIMPLE providers use **heap size**.

CIMOM vs. provider size

- The object size of the CIMOM is a fixed cost.
- The cost of the provider is a variable cost (it increases as you add or enhance providers).

Faster

- Use of first-class objects reduces overhead by one order of magnitude (no longer necessary to search property lists).

Reliable

Due to first class objects:

- more errors can be detected at compile time.
- complexity is substantially reduced.

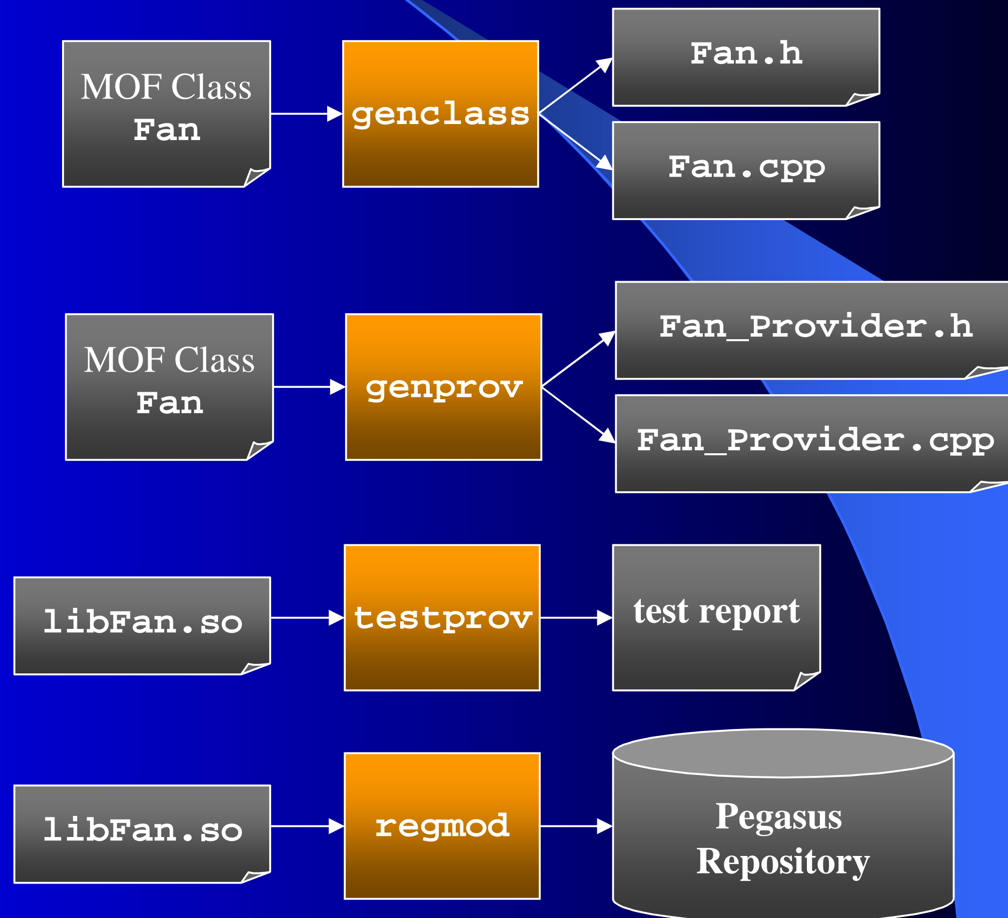
Interoperable

- CIMPLE was designed to work with any CIMOM (or environment).
- CIMPLE provider can be loaded by Pegasus today.

Developing the provider

CIMPLE provider development

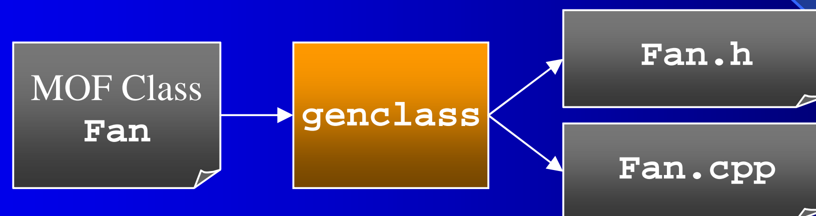
1. Define MOF class.
2. Generate class (**`genclass`**).
3. Generate provider skeleton (**`genprov`**).
4. Implement operations.
5. Pre-test provider (**`testprov`**).
6. Register provider with Pegasus.



Define the MOF class

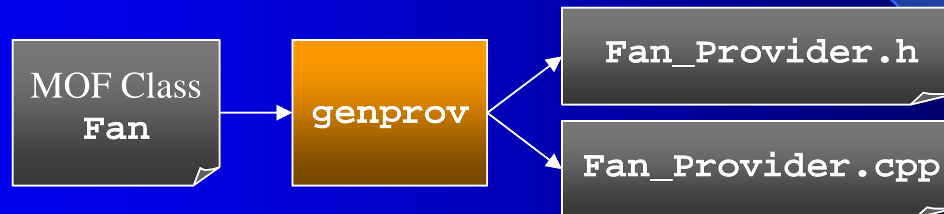
```
// repository.mof

class Fan
{
    [key] string DeviceID;
    uint64 Speed;
    uint64 DesiredSpeed;
    uint32 SetSpeed([in] uint64 DesiredSpeed);
};
```

Generate the class




```
% genclass Fan
Created Fan.h
Created Fan.cpp
% _
```

Generate provider skeleton




```
% genprov Fan
Created Fan_Provider.h
Created Fan_Provider.cpp
% _
```

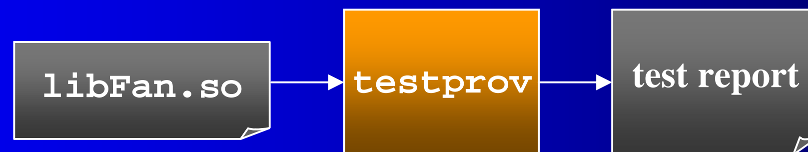
get_instance method

```
// Fan_Provider.cpp:

Get_Instance_Status Fan_Provider::get_instance(
    const Fan* model,
    Fan*& instance)
{
    instance = model->clone();

    // Only one fan in this system.

    if (inst->DeviceID.value == “FAN01”)
    {
        inst->Speed.value = _get_fan_speed(1);
        inst->DesiredSpeed.value = 0;
        return GET_INSTANCE_OK;
    }

    return GET_INSTANCE_NOT_FOUND;
}
```

SetSpeed method

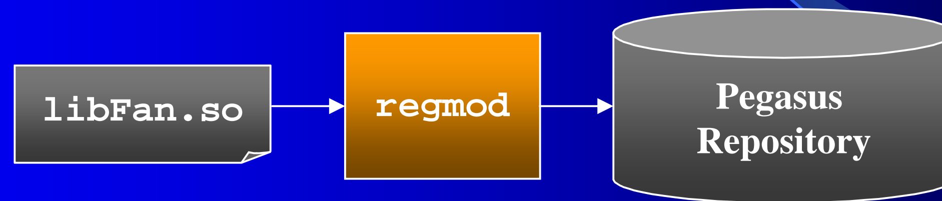
```
// Fan_Provider.cpp:

Invoke_Method_Status Fan_Provider::SetSpeed(
    const Fan* inst,
    const Value<uint64>& DesiredSpeed,
    Value<uint32>& return_value)
{
    // Only one fan in this system.

    if (inst->DeviceID == "FAN01")
    {
        _set_desired_speed(1, DesiredSpeed);
        return_value.value = _get_fan_speed(1);
        return INVOKE_METHOD_OK;
    }

    return INVOKE_METHOD_FAILED;
}

// Note: Pegasus version of SetSpeed() is 87 lines of code.
```

Pre-test the provider




```
% testprov libcmplfan.so
% _
```

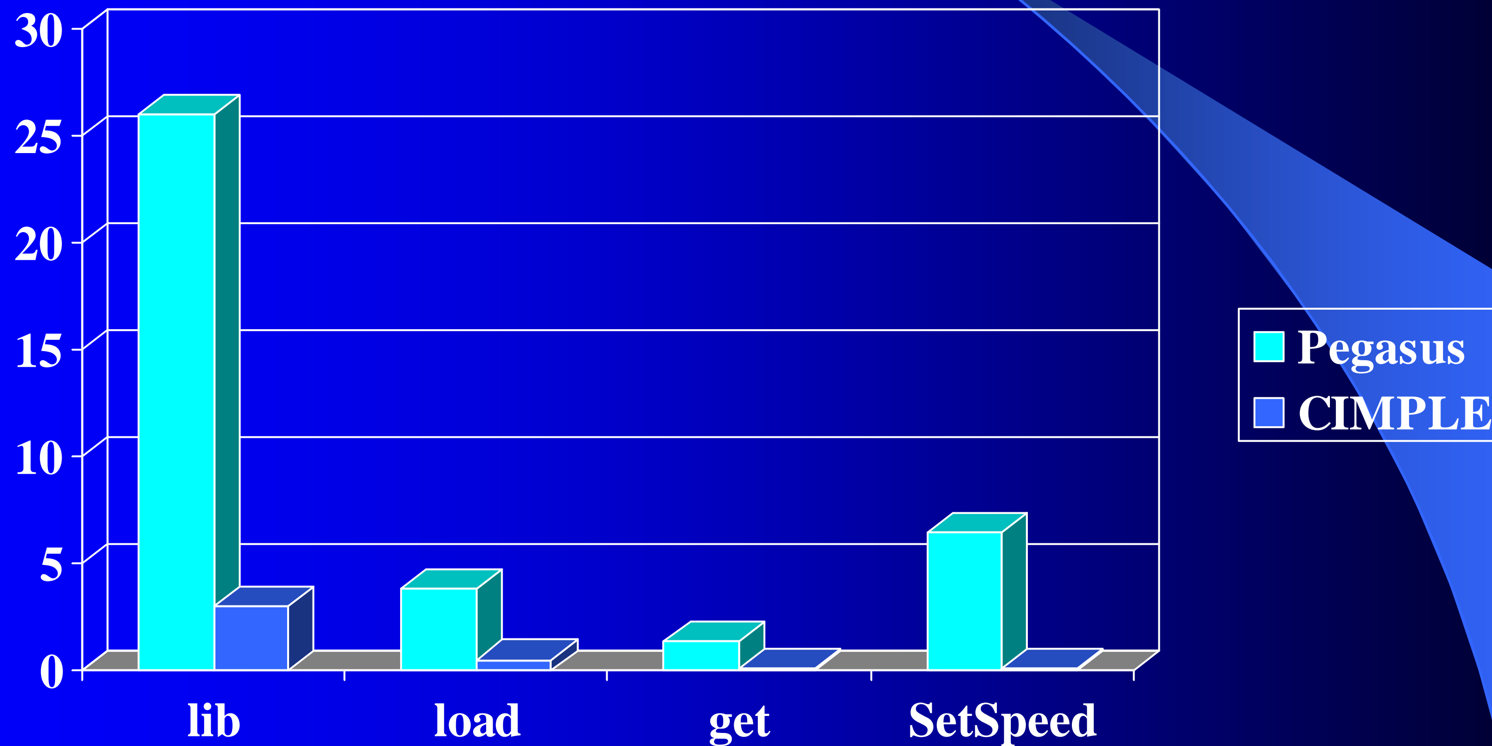
Register module with Pegasus




```
% regmod libcmplfan.so
% _
```

Provider size comparison

(kilobytes)



Note: savings increases with the complexity of the provider

Conclusion

Status

Complete:
- Instance providers
- Method providers
- Association providers
- Indication providers
- CMPI adapter*

Work in progress:
- Query providers
- Access control

Obtaining CIMPLE

- CIMPLE can be downloaded from **http://www.cimple.org**.

Contact us

- **Karl Schopmeyer – k.schopmeyer@inovadevelopment.com**
- **Mike Brasher – m.brasher@inovadevelopment.com**