

KonkretCMPI

Michael E. Brasher
June 6, 2008

Copyright © 2008 by Michael Brasher

Table of Contents

1	Introduction	1
2	Developing an Instance Provider	2
2.1	Generating the Sources.....	2
2.2	Manipulating Generated Instances.....	2
2.3	Implementing <code>WidgetEnumInstances</code>	5
2.4	Implementing <code>WidgetGetInstance</code>	6
2.5	Implementing <code>WidgetEnumerateInstances</code>	8
3	Developing a Method Provider	9
3.1	Examining <code>WidgetInvokeMethod</code>	9
3.2	Implementing <code>Widget.Add</code>	9
4	Developing an Association Provider	11
4.1	Generating the Sources.....	11
4.2	Implementing <code>GadgetEnumInstances</code>	11
4.3	Examining <code>GadgetAssociators</code>	13
5	Developing an Indication Provider	14
Appendix A	<code>Widget.h</code> Listing	15
Appendix B	<code>WidgetProvider.c</code> Listing	25
Appendix C	<code>Gadget.h</code> Listing	28
Appendix D	<code>GadgetProvider.c</code> Listing	37

1 Introduction

KonkretCMPI is a tool for rapidly building CMPI providers in the C programming language. It is dedicated to improving the productivity of CMPI developers while satisfying the following objectives.

- Builds on CMPI rather than introducing a new provider interface.
- Requires no other programming language other than C.
- Introduces no runtime library dependencies.
- Produces providers with a very small footprint.
- Can be used with existing CMPI providers.

KonkretCMPI improves productivity in three ways.

- By generating concrete type-safe C interfaces for manipulating CIM elements.
- By providing default implementations for many provider operations.
- By supplying convenience functions for working with CMPI.

The following chapters illustrate all of these with examples. The next chapter shows how to develop a simple instance provider with KonkretCMPI.

2 Developing an Instance Provider

This chapter shows how to develop an instance provider for the following MOF class.

```
class KC_Widget
{
    [Key] uint32 Id;
    string Color;
    uint32 Size;

    [Static]
    uint32 Add([In] uint32 X, [In] uint32 Y);
};
```

The following sections develop the provider step by step.

2.1 Generating the Sources

First we generate the source code from the MOF class definition as follows.

```
$ konkret -s KC_Widget -m Widget.mof KC_Widget=Widget
Created Widget.h
Created WidgetProvider.c
```

This command creates two files (omit ‘-s’ if you only want `Widget.h`).

- `Widget.h` – contains the `Widget` class interfaces.
- `WidgetProvider.c` – contains the `Widget` provider skeleton.

Complete listings of these files are contained in Appendix A and Appendix B.

2.2 Manipulating Generated Instances

Before implementing the provider operations we must show how to manipulate instances of generated classes. `Widget.h` contains three sorts of definitions.

- The `Widget` structure and associated functions.
- The `WidgetRef` structure and associated functions.
- Extrinsic method prototypes and a function for invoking them.

The `Widget` structure represents the `Widget` MOF class shown earlier and contains the same properties. The `WidgetRef` struct contains only the key properties of the `Widget` MOF class and is used to represent `Widget` references (object paths).

We begin by examining the generated `Widget` structure shown below.

```
typedef struct _Widget
{
    KBase __base;
    /* KC_Widget features */
    const KString Id;
    const KString Color;
    const KUInt32 Size;
}
```

```
Widget;
```

All generated structures contain the private `__base` field followed by one or more fields that either represent CIM properties or CIM references. Properties always have one of the following types.

- `KBoolean`
- `KUInt8`
- `KSint8`
- `KUInt16`
- `KSint16`
- `KUInt32`
- `KSint32`
- `KUInt64`
- `KSint64`
- `KReal32`
- `KReal64`
- `KChar16`
- `KString`
- `KDateTime`

Each of these is a structure with the following fields:

- `exists` – indicates whether property exists.
- `null` – indicates whether property is null.
- `value` – contains the current value of the property.

Take for example the `KUInt32` structure defined below.

```
typedef struct _KUInt32
{
    CMPIUInt32 exists;
    CMPIUInt32 null;
    CMPIUInt32 value;
}
KUInt32;
```

A non-zero `KUInt32.exists` field indicates that the field exists or is present. Recall that CMPI providers can form partial instances with only a subset of the properties specified. A non-zero `KUInt32.null` field indicates that the given field is null. Recall that CIM properties may be null (i.e., no value is present). The `KUInt32.value` field, as you probably guessed contains the value of the property itself. Setting the state of these three fields is tricky so KonkretCMPI generates special modifier functions for setting these safely. The user must use the modifiers and must never modify these fields directly. These modifiers are discussed shortly.

All other type structures define the three fields described above. The `KString` structure, shown below, defines an additional `chars` field for convenience.

```

typedef struct _KString
{
    CMPIUint32 exists;
    CMPIUint32 null;
    CMPIString* value;
    const char* chars;
}
KString;

```

The `chars` field always refers to the C-string representation of the `CMPIString` value field. Again, users must only modify these fields indirectly through the generated modifier functions discussed below.

Now we show how to manipulate generated structures. The following fragment declares and initializes an instance of the `Widget` class.

```

const CMPIBroker* broker;
const char* nameSpace;
Widget w;
...
Widget_Init(&w, broker, nameSpace);

```

`Widget_Init` initializes the `Widget` instance and clears the `exists` flag in every property in the structure. So by default, none of the properties of an instance exist. An instance may be initialized from a `CMPIInstance` with `Widget_InitFromInstance` or from a `CMPIObjectPath` with `Widget_InitFromObjectPath`.

The following snippet sets the values of the three properties of the `Widget` instance.

```

Widget_Set_Id(&w, "1001");
Widget_Set_Color(&w, "Red");
Widget_Set_Size(&w, 1);

```

For each property, the `exists` flag is set to non-zero (one), the `null` flag is set to zero, and the value is set as indicated. `Widget_Set_Color` sets the string `Color` property from a C-String. To set it from a `CMPIString` use `Widget_SetString_Color` instead.

KonkretCMPI also generates modifiers for clearing the `exists` field and setting the `null` field. For example, the following snippet clears the `Widget.Color` field and nulls the `Widget.Size` field.

```

Widget_Clr_Id(&w);
Widget_Null_Size(&w);

```

Although properties should only be modified indirectly with generated modifiers, they may be read directly. For example, the following snippet prints the fields of the `Widget.Color` property created above.

```

printf("%u %u %s\n", w.Color.exists, w.Color.null, w.Color.chars);

```

KonkretCMPI also generates print functions for each generated class. For example, the following snippet prints a `Widget` instance to standard output.

```

Widget_Print(&w, stdout);

```

Functions are generated for converting any instance to a `CMPIInstance` or a `CMPIObjectPath`. For example, the following converts the `Widget` instance created above to a `CMPIInstance`.

```

    CMPIInstance* instance;
    CMPIStatus* status;
    ...
    instance = Widget_ToInstance(&w, &status);

```

Everything we have said about the `Widget` structure and its associated functions also applies to the `WidgetRef` structure. We will see examples that employ the `WidgetRef` structure later.

The main advantage of `KonkretCMPI` is allowing developers to work in the **concrete domain**. As we will see, providers written with `KonkretCMPI` work with almost exclusively with concrete instances, only encountering `CMPIInstances` and `CMPIObjectPaths` in the incoming and outgoing boundaries of the provider interface. This schema affords the greatest degree of convenience and type-safety without introducing a new provider interface. Hence, `KonkretCMPI` augments `CMPI` rather than replacing it.

2.3 Implementing `WidgetEnumInstances`

We now consider how to implement the `WidgetEnumInstances` skeleton that was generated above.

```

    CMPIStatus WidgetEnumInstances(
        CMPIInstanceMI* mi,
        const CMPIContext* context,
        const CMPIResult* result,
        const CMPIObjectPath* cop,
        const char** properties)
    {
        CMReturn(CMPI_RC_OK);
    }

```

Our implementation, shown below, yields three instances.

```

    CMPIStatus WidgetEnumInstances(
        CMPIInstanceMI* mi,
        const CMPIContext* context,
        const CMPIResult* result,
        const CMPIObjectPath* cop,
        const char** properties)
    {
        Widget w;

        /* Widget.Id="1001" */
        Widget_Init(&w, _broker, KNamespace(cop));
        Widget_Set_Id(&w, "1001");
        Widget_Set_Color(&w, "Red");
        Widget_Set_Size(&w, 1);
        KReturnInstance(result, w);

        /* Widget.Id="1002" */
        Widget_Init(&w, _broker, KNamespace(cop));

```

```

Widget_Set_Id(&w, "1002");
Widget_Set_Color(&w, "Green");
Widget_Set_Size(&w, 2);
KReturnInstance(result, w);

/* Widget.Id="1003" */
Widget_Init(&w, _broker, KNameSpace(cop));
Widget_Set_Id(&w, "1003");
Widget_Set_Color(&w, "Blue");
Widget_Set_Size(&w, 3);
KReturnInstance(result, w);

CMReturn(CMPI_RC_OK);
}

```

This example creates and delivers three instances. There are two convenience functions we have not discussed: the `KNameSpace` and `KReturnInstance`. `KNameSpace` returns the namespace from a `CMPIObjectPath` as a C-string. `KReturnInstance` converts a `Widget` into a `CMPIInstance` and returns it.

2.4 Implementing `WidgetGetInstance`

We now consider how to implement `WidgetGetInstance`. The skeleton generated above follows.

```

CMPIStatus WidgetGetInstance(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    const CMPIResult* result,
    const CMPIObjectPath* cop,
    const char** properties)
{
    return KDefaultGetInstance(_broker, mi, cc, result, cop, properties);
}

```

This is already a working implementation. `KDefaultGetInstance` implements this function in terms of `WidgetEnumInstances`. That is, it calls `WidgetEnumInstances` searching for an instance that matches the `cop` parameter. If found, it delivers that instance to the server and returns `CMPI_RC_OK`.

Alternatively, we can implement `WidgetGetInstance` explicitly as shown below.

```

CMPIStatus WidgetGetInstance(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    const CMPIResult* result,
    const CMPIObjectPath* cop,
    const char** properties)
{
    WidgetRef wr;
    Widget w;

```

```

WidgetRef_InitFromObjectPath(&wr, _broker, cop);

if (!wr.Id.exists || wr.Id.null)
    CMReturn(CMPI_RC_ERR_FAILED);

if (strcmp(wr.Id.chars, "1001") == 0)
{
    Widget_Init(&w, _broker, KNameSpace(cop));
    Widget_Set_Id(&w, "1001");
    Widget_Set_Color(&w, "Red");
    Widget_Set_Size(&w, 1);
    KReturnInstance(result, w);
    CMReturn(CMPI_RC_OK);
}
else if (strcmp(wr.Id.chars, "1002") == 0)
{
    Widget_Init(&w, _broker, KNameSpace(cop));
    Widget_Set_Id(&w, "1002");
    Widget_Set_Color(&w, "Green");
    Widget_Set_Size(&w, 2);
    KReturnInstance(result, w);
    CMReturn(CMPI_RC_OK);
}
else if (strcmp(wr.Id.chars, "1003") == 0)
{
    Widget_Init(&w, _broker, KNameSpace(cop));
    Widget_Set_Id(&w, "1003");
    Widget_Set_Color(&w, "Blue");
    Widget_Set_Size(&w, 3);
    KReturnInstance(result, w);
    CMReturn(CMPI_RC_OK);
}

CMReturn(CMPI_RC_ERR_NOT_FOUND);
}

```

This implementation has the advantage of being faster than the previous one since it only has to create a single instance (rather than all of them). This implementation begins by converting the `cop` parameter to a concrete `WidgetRef` reference. It then errors out if the `WidgetRef.Id` field is absent or null. Next, it checks the `WidgetRef.Id.chars` field to determine which instance to produce. If it matches a known instance, it create and returns it. Otherwise, it returns `CMPI_RC_ERR_NOT_FOUND`.

By now you probably have noticed that learning KonkretCMPI is mostly a matter of learning to work with concrete references and instances. Everything else is familiar to a CMPI developer.

2.5 Implementing WidgetEnumerateInstances

We now examine the `WidgetEnumerateInstances` generated earlier and shown below.

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

The `KDefaultEnumerateInstanceNames` provides a working implementation, that implements `WidgetEnumInstanceNames` in terms of `WidgetEnumerateInstances`. Implementing this more efficiently is left as an exercise.

3 Developing a Method Provider

This chapter shows how to implement a method provider by extending the `Widget` instances provider introduced in the previous chapter. Recall the MOF `Widget` definition.

```
class KC_Widget
{
    [Key] uint32 Id;
    string Color;
    uint32 Size;

    [Static]
    uint32 Add([In] uint32 X, [In] uint32 Y);
};
```

We will show how to implement the `Widget.Add` extrinsic method in the following sections.

3.1 Examining `WidgetInvokeMethod`

KonkretCMPI automatically generates the following implementation for the `WidgetInvokeMethod` function. This function requires no change and handles dispatching of all extrinsic methods by calling the generated `Widget_InvokeMethod` function defined in `Widget.h`.

```
CMPIStatus WidgetInvokeMethod(
    CMPIMethodMI* mi,
    const CMPIContext* context,
    const CMPIResult* result,
    const CMPIObjectPath* cop,
    const char* meth,
    const CMPIArgs* in,
    CMPIArgs* out)
{
    return Widget_InvokeMethod(
        _broker, mi, context, result, cop, meth, in, out);
}
```

If you examine `Widget_InvokeMethod`, you will see that it dispatches the `Add` method to the `Widget_Add` skeleton contained in `WidgetProvider.c`. The implementation of this skeleton is discussed in the next section.

3.2 Implementing `Widget.Add`

KonkretCMPI generated the following skeleton for the `Widget.Add` method.

```
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;
}

```

This function sets the output status to `ERR_NOT_SUPPORTED` and returns the result, which is initialized to `KUINT32_INIT`. `KUINT32_INIT` is the default value for `KUInt32`. It sets its fields as follows.

- `Kuint32.exists` – to zero.
- `Kuint32.null` – to zero.
- `Kuint32.value` – to zero.

We provide the following implementation that checks that the input parameters (X and Y) "exist" and that they are non-null. It then sets the result as the sum of these and returns success.

```

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;
}

```

Note the use of `KUInt32_Set`, which is similar to property set functions but works on basic types.

4 Developing an Association Provider

This chapter shows how to develop an association provider for the following MOF class.

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

The following sections develop the provider step by step.

4.1 Generating the Sources

First we generate the source code from the MOF class definition as follows.

```
$ konkret -s KC_Gadget -m Widget.mof -m Gadget.mof KC_Gadget=Gadget KC_Widget=Widget
Created Gadget.h
Created GadgetProvider.c
Created Widget.h
```

This command creates three files (omit ‘-s’ if you only want `Gadget.h` and `Widget.h`).

- `Widget.h` – contains the `Widget` class interfaces.
- `Gadget.h` – contains the `Gadget` class interfaces.
- `GadgetProvider.c` – contains the `Gadget` provider skeleton.

Complete listings of `Gadget.h` and `GadgetProvider.c` are contained in Appendix C and Appendix D.

4.2 Implementing `GadgetEnumInstances`

The generated provider skeleton is a working provider that happens to provide zero instances. By implementing just `GadgetEnumInstances` the association provider is a complete read-only association provider. Complete implementations of the following provider methods were generated with the skeleton (although you might decide to provide custom implementations to improve performance).

- `GadgetEnumerateInstanceNames` – implemented via `KDefaultEnumerateInstanceNames`.
- `GadgetGetInstance` – implemented via `KDefaultGetInstance`.
- `GadgetAssociators` – implemented via `KDefaultAssociators`.
- `GadgetAssociatorNames` – implemented via `KDefaultAssociatorNames`.
- `GadgetReferences` – implemented via `KDefaultReferences`.
- `GadgetReferenceNames` – implemented via `KDefaultReferenceNames`.

Here is our implementation of `GadgetEnumInstances`.

```
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;

    /* First Gadget */
    {
        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);
    }

    /* Second Gadget */
    {
        WidgetRef_Init(&left, _broker, ns);
        WidgetRef_Set_Id(&left, "1001");

        WidgetRef_Init(&right, _broker, ns);
        WidgetRef_Set_Id(&right, "1003");

        Gadget_Init(&g, _broker, ns);
        Gadget_Set_Left(&g, &left);
        Gadget_Set_Right(&g, &right);
        KReturnInstance(result, g);
    }

    CMReturn(CMPI_RC_OK);
}

```

This implementation uses two generated class interfaces: `WidgetRef` and `Gadget`. It produces two `Gadget` association instances, whose objects paths are:

- `KC_Gadget.Left="KC_Widget.Id=\"1001\",Right="KC_Widget.Id=\"1002\""`
- `KC_Gadget.Left="KC_Widget.Id=\"1001\",Right="KC_Widget.Id=\"1003\""`

The other implementation details should be familiar after reading about the `WidgetEnumerateInstances` implementation.

4.3 Examining GadgetAssociators

We now examine the generated `GadgetAssociators` implementation, shown below.

```
CMPIStatus GadgetAssociators(  
    CMPIAssociationMI* mi,  
    const CMPIContext* cc,  
    const CMPIResult* cr,  
    const CMPIObjectPath* cop,  
    const char* assocClass,  
    const char* resultClass,  
    const char* role,  
    const char* resultRole,  
    const char** properties)  
{  
    if (!assocClass)  
        assocClass = "KC_Gadget";  
  
    return KDefaultAssociators(_broker, mi, cc, cr, cop, assocClass,  
        resultClass, role, resultRole, properties);  
}
```

This operation is implemented through `KDefaultAssociators`, which uses `GadgetEnumInstanceNames` and `CBGetInstance` to satisfy the request. Note this functions sets `assocClass` to "KC_Gadget" if null. This is required since `KDefaultAssociators` fails if `assocClass` is null.

5 Developing an Indication Provider

KonkretCMPI generates indication provider skeletons through processes already familiar to the reader. Indication providers are implemented using the concrete concepts already introduced. Therefore, we do not cover development of indication provider in this document.

Appendix A Widget.h Listing

```

/*
**=====
**
** CAUTION: This file generated by KonkretCMPI. Please do not edit.
**
**=====
*/

#ifndef _konkrete_Widget_h
#define _konkrete_Widget_h

#include <konkret/konkret.h>

/*
**=====
**
** struct WidgetRef
**
**=====
*/

/* classname=KC_Widget */
typedef struct _WidgetRef
{
    KBase __base;
    /* KC_Widget features */
    const KString Id;
}
WidgetRef;

static const unsigned char __WidgetRef_sig[] =
{
    0x09,0x4b,0x43,0x5f,0x57,0x69,0x64,0x67,0x65,0x74,0x00,0x01,0x4c,0x02,0x49,
    0x64,0x00,
};

KINLINE void WidgetRef_Init(
    WidgetRef* self,
    const CMPIBroker* cb,
    const char* ns)
{
    const unsigned char* sig = __WidgetRef_sig;
    KBase_Init(&self->__base, cb, sizeof(*self), sig, ns);
}

```

```
KINLINE CMPIStatus WidgetRef_InitFromInstance(
    WidgetRef* self,
    const CMPIBroker* cb,
    const CMPIInstance* x)
{
    WidgetRef_Init(self, cb, NULL);
    return KBase_FromInstance(&self->__base, x);
}

KINLINE CMPIStatus WidgetRef_InitFromObjectPath(
    WidgetRef* self,
    const CMPIBroker* cb,
    const CMPIObjectPath* x)
{
    WidgetRef_Init(self, cb, NULL);
    return KBase_FromObjectPath(&self->__base, x);
}

KINLINE void WidgetRef_Print(
    const WidgetRef* self,
    FILE* os)
{
    KBase_Print(os, &self->__base, 'r');
}

KINLINE CMPIInstance* WidgetRef_ToInstance(
    const WidgetRef* self,
    CMPIStatus* status)
{
    return KBase_ToInstance(&self->__base, status);
}

KINLINE CMPIObjectPath* WidgetRef_ToObjectPath(
    const WidgetRef* self,
    CMPIStatus* status)
{
    return KBase_ToObjectPath(&self->__base, status);
}

KINLINE const char* WidgetRef_NameSpace(
    WidgetRef* self)
{
    if (self && self->__base.magic == KMAGIC)
        return self->__base.ns ? KChars(self->__base.ns) : NULL;
    return NULL;
}
```

```

KINLINE void WidgetRef_SetString_Id(
    WidgetRef* self,
    CMPIStrng* x)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KString* field = (KString*)&self->Id;
        KString_SetString(field, x);
    }
}

KINLINE void WidgetRef_Set_Id(
    WidgetRef* self,
    const char* s)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KString* field = (KString*)&self->Id;
        KString_Set(field, self->__base.cb, s);
    }
}

KINLINE void WidgetRef_Null_Id(
    WidgetRef* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KString* field = (KString*)&self->Id;
        KString_Null(field);
    }
}

KINLINE void WidgetRef_Clr_Id(
    WidgetRef* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KString* field = (KString*)&self->Id;
        KString_Clr(field);
    }
}

/*
**=====
**
** struct Widget
**

```

```

**=====
*/

/* classname=KC_Widget */
typedef struct _Widget
{
    KBase __base;
    /* KC_Widget features */
    const KString Id;
    const KString Color;
    const KUInt32 Size;
}
Widget;

static const unsigned char __Widget_sig[] =
{
    0x09,0x4b,0x43,0x5f,0x57,0x69,0x64,0x67,0x65,0x74,0x00,0x03,0x4c,0x02,0x49,
    0x64,0x00,0x0c,0x05,0x43,0x6f,0x6c,0x6f,0x72,0x00,0x05,0x04,0x53,0x69,0x7a,
    0x65,0x00,
};

KINLINE void Widget_Init(
    Widget* self,
    const CMPIBroker* cb,
    const char* ns)
{
    const unsigned char* sig = __Widget_sig;
    KBase_Init(&self->__base, cb, sizeof(*self), sig, ns);
}

KINLINE CMPIStatus Widget_InitFromInstance(
    Widget* self,
    const CMPIBroker* cb,
    const CMPIInstance* x)
{
    Widget_Init(self, cb, NULL);
    return KBase_FromInstance(&self->__base, x);
}

KINLINE CMPIStatus Widget_InitFromObjectPath(
    Widget* self,
    const CMPIBroker* cb,
    const CMPIObjectPath* x)
{
    Widget_Init(self, cb, NULL);
    return KBase_FromObjectPath(&self->__base, x);
}

```

```
KINLINE void Widget_Print(  
    const Widget* self,  
    FILE* os)  
{  
    KBase_Print(os, &self->__base, 'i');  
}  
  
KINLINE CMPIInstance* Widget_ToInstance(  
    const Widget* self,  
    CMPIStatus* status)  
{  
    return KBase_ToInstance(&self->__base, status);  
}  
  
KINLINE CMPIObjectPath* Widget_ToObjectPath(  
    const Widget* self,  
    CMPIStatus* status)  
{  
    return KBase_ToObjectPath(&self->__base, status);  
}  
  
KINLINE const char* Widget_NameSpace(  
    Widget* self)  
{  
    if (self && self->__base.magic == KMAGIC)  
        return self->__base.ns ? KChars(self->__base.ns) : NULL;  
    return NULL;  
}  
  
KINLINE void Widget_SetString_Id(  
    Widget* self,  
    CMPIString* x)  
{  
    if (self && self->__base.magic == KMAGIC)  
    {  
        KString* field = (KString*)&self->Id;  
        KString_SetString(field, x);  
    }  
}  
  
KINLINE void Widget_Set_Id(  
    Widget* self,  
    const char* s)  
{  
    if (self && self->__base.magic == KMAGIC)  
    {
```

```
        KString* field = (KString*)&self->Id;
        KString_Set(field, self->__base.cb, s);
    }
}

KINLINE void Widget_Null_Id(
    Widget* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KString* field = (KString*)&self->Id;
        KString_Null(field);
    }
}

KINLINE void Widget_Clr_Id(
    Widget* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KString* field = (KString*)&self->Id;
        KString_Clr(field);
    }
}

KINLINE void Widget_SetString_Color(
    Widget* self,
    CMPIString* x)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KString* field = (KString*)&self->Color;
        KString_SetString(field, x);
    }
}

KINLINE void Widget_Set_Color(
    Widget* self,
    const char* s)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KString* field = (KString*)&self->Color;
        KString_Set(field, self->__base.cb, s);
    }
}
```

```
KINLINE void Widget_Null_Color(
    Widget* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KString* field = (KString*)&self->Color;
        KString_Null(field);
    }
}

KINLINE void Widget_Clr_Color(
    Widget* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KString* field = (KString*)&self->Color;
        KString_Clr(field);
    }
}

KINLINE void Widget_Set_Size(
    Widget* self,
    CMPIUint32 x)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KUint32* field = (KUint32*)&self->Size;
        KUint32_Set(field, x);
    }
}

KINLINE void Widget_Null_Size(
    Widget* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KUint32* field = (KUint32*)&self->Size;
        KUint32_Null(field);
    }
}

KINLINE void Widget_Clr_Size(
    Widget* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KUint32* field = (KUint32*)&self->Size;
```

```

        KUint32_Clr(field);
    }
}

/* classname=KC_Widget */
typedef struct _Widget_Add_Args
{
    KBase __base;
    /* IN */
    KUint32 X;
    /* IN */
    KUint32 Y;
}
Widget_Add_Args;

static const unsigned char __Widget_Add_Args_sig[] =
{
    0x03,0x41,0x64,0x64,0x00,0x02,0x25,0x01,0x58,0x00,0x25,0x01,0x59,0x00,
};

KINLINE void Widget_Add_Args_Init(
    Widget_Add_Args* self,
    const CMPIBroker* cb)
{
    const unsigned char* sig = __Widget_Add_Args_sig;
    KBase_Init(&self->__base, cb, sizeof(*self), sig, NULL);
}

KINLINE CMPIStatus Widget_Add_Args_InitFromArgs(
    Widget_Add_Args* self,
    const CMPIBroker* cb,
    const CMPIArgs* x,
    CMPIBoolean in,
    CMPIBoolean out)
{
    Widget_Add_Args_Init(self, cb);
    return KBase_FromArgs(&self->__base, x, in, out);
}

KINLINE CMPIArgs* Widget_Add_Args_ToArgs(
    const Widget_Add_Args* self,
    CMPIBoolean in,
    CMPIBoolean out,
    CMPIStatus* status)
{
    return KBase_ToArgs(&self->__base, in, out, status);
}

```

```

KINLINE CMPIStatus Widget_Add_Args_SetArgs(
    const Widget_Add_Args* self,
    CMPIBoolean in,
    CMPIBoolean out,
    CMPIArgs* ca)
{
    return KBase_SetToArgs(&self->__base, in, out, ca);
}

KINLINE void Widget_Add_Args_Print(
    const Widget_Add_Args* self,
    FILE* os)
{
    KBase_Print(os, &self->__base, 'a');
}

/*
**=====
**
** Widget methods
**
**=====
*/

extern KUInt32 Widget_Add(
    const CMPIBroker* cb,
    CMPIMethodMI* mi,
    const CMPIContext* context,
    const KUInt32* X,
    const KUInt32* Y,
    CMPIStatus* status);

KINLINE CMPIStatus Widget_InvokeMethod(
    const CMPIBroker* cb,
    CMPIMethodMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const char* meth,
    const CMPIArgs* in,
    CMPIArgs* out)
{
    WidgetRef self;

    KReturnIf(WidgetRef_InitFromObjectPath(&self, cb, cop));
}

```

```
if (strcasecmp(meth, "Add") == 0)
{
    CMPIStatus st = KSTATUS_INIT;
    Widget_Add_Args args;
    KUint32 r;

    KReturnIf(Widget_Add_Args_InitFromArgs(
        &args, cb, in, 1, 0));

    r = Widget_Add(
        cb,
        mi,
        cc,
        &args.X,
        &args.Y,
        &st);

    if (!KOkay(st))
        return st;

    if (!r.exists)
        KReturn(ERR_FAILED);

    KReturnIf(Widget_Add_Args_SetArgs(
        &args, 0, 1, out));
    KReturnUint32Data(cr, &r);
    CMReturnDone(cr);

    KReturn(OK);
}

KReturn(ERR_METHOD_NOT_FOUND);
}

#endif /* _konkrete_Widget_h */
```

Appendix B WidgetProvider.c Listing

```
#include <konkret/konkret.h>
#include "Widget.h"

static const CMPIBroker* _broker = NULL;

static void WidgetInitialize()
{
}

CMPIStatus WidgetCleanup(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    CMPIBoolean term)
{
    CMReturn(CMPI_RC_OK);
}

CMPIStatus WidgetEnumInstanceNames(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop)
{
    return KDefaultEnumerateInstanceNames(
        _broker, mi, cc, cr, cop);
}

CMPIStatus WidgetEnumInstances(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const char** properties)
{
    CMReturn(CMPI_RC_OK);
}

CMPIStatus WidgetGetInstance(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const char** properties)
{
    return KDefaultGetInstance(
```

```
        _broker, mi, cc, cr, cop, properties);
}

CMPIDStatus WidgetCreateInstance(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const CMPIInstance* ci)
{
    CMReturn(CMPI_RC_ERR_NOT_SUPPORTED);
}

CMPIDStatus WidgetModifyInstance(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const CMPIInstance* ci,
    const char** properties)
{
    CMReturn(CMPI_RC_ERR_NOT_SUPPORTED);
}

CMPIDStatus WidgetDeleteInstance(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop)
{
    CMReturn(CMPI_RC_ERR_NOT_SUPPORTED);
}

CMPIDStatus WidgetExecQuery(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const char* lang,
    const char* query)
{
    CMReturn(CMPI_RC_ERR_NOT_SUPPORTED);
}

CMInstanceMIStub(Widget, Widget, _broker, WidgetInitialize())

CMPIDStatus WidgetMethodCleanup(
```

```
    CMPIMethodMI* mi,
    const CMPIContext* cc,
    CMPIBoolean term)
{
    CMReturn(CMPI_RC_OK);
}

CMPIStatus WidgetInvokeMethod(
    CMPIMethodMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const char* meth,
    const CMPIArgs* in,
    CMPIArgs* out)
{
    return Widget_InvokeMethod(
        _broker, mi, cc, cr, cop, meth, in, out);
}

CMMethodMIStub(
    Widget,
    Widget,
    _broker,
    WidgetInitialize())

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;
}
```

Appendix C Gadget.h Listing

```

/*
**=====
**
** CAUTION: This file generated by KonkretCMPI. Please do not edit.
**
**=====
*/

#ifndef _konkrete_Gadget_h
#define _konkrete_Gadget_h

#include <konkret/konkret.h>
#include "Widget.h"

/*
**=====
**
** struct GadgetRef
**
**=====
*/

/* classname=KC_Gadget */
typedef struct _GadgetRef
{
    KBase __base;
    /* KC_Gadget features */
    const KRef Left; /* Widget */
    const KRef Right; /* Widget */
}
GadgetRef;

static const unsigned char __GadgetRef_sig[] =
{
    0x09,0x4b,0x43,0x5f,0x47,0x61,0x64,0x67,0x65,0x74,0x00,0x02,0x4e,0x04,0x4c,
    0x65,0x66,0x74,0x00,0x4e,0x05,0x52,0x69,0x67,0x68,0x74,0x00,
};

KINLINE void GadgetRef_Init(
    GadgetRef* self,
    const CMPIBroker* cb,
    const char* ns)
{
    const unsigned char* sig = __GadgetRef_sig;
    KBase_Init(&self->__base, cb, sizeof(*self), sig, ns);
}

```

```
    ((KRef*)&self->Left)->__sig = __Widget_sig;
    ((KRef*)&self->Right)->__sig = __Widget_sig;
}

KINLINE CMPIStatus GadgetRef_InitFromInstance(
    GadgetRef* self,
    const CMPIBroker* cb,
    const CMPIInstance* x)
{
    GadgetRef_Init(self, cb, NULL);
    return KBase_FromInstance(&self->__base, x);
}

KINLINE CMPIStatus GadgetRef_InitFromObjectPath(
    GadgetRef* self,
    const CMPIBroker* cb,
    const CMPIObjectPath* x)
{
    GadgetRef_Init(self, cb, NULL);
    return KBase_FromObjectPath(&self->__base, x);
}

KINLINE void GadgetRef_Print(
    const GadgetRef* self,
    FILE* os)
{
    KBase_Print(os, &self->__base, 'r');
}

KINLINE CMPIInstance* GadgetRef_ToInstance(
    const GadgetRef* self,
    CMPIStatus* status)
{
    return KBase_ToInstance(&self->__base, status);
}

KINLINE CMPIObjectPath* GadgetRef_ToObjectPath(
    const GadgetRef* self,
    CMPIStatus* status)
{
    return KBase_ToObjectPath(&self->__base, status);
}

KINLINE const char* GadgetRef_NameSpace(
    GadgetRef* self)
{
    if (self && self->__base.magic == KMAGIC)
```

```

        return self->__base.ns ? KChars(self->__base.ns) : NULL;
    return NULL;
}

KINLINE void GadgetRef_SetObjectPath_Left(
    GadgetRef* self,
    const CMPIObjectPath* x)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Left;
        KRef_SetObjectPath(field, x);
    }
}

KINLINE CMPIStatus GadgetRef_Set_Left(
    GadgetRef* self,
    const WidgetRef* x)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Left;
        return KRef_Set(field, &x->__base);
    }
    CMReturn(CMPI_RC_ERR_FAILED);
}

KINLINE void GadgetRef_Null_Left(
    GadgetRef* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Left;
        KRef_Null(field);
    }
}

KINLINE void GadgetRef_Clr_Left(
    GadgetRef* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Left;
        KRef_Clr(field);
    }
}

```

```

KINLINE void GadgetRef_SetObjectPath_Right(
    GadgetRef* self,
    const CMPIObjectPath* x)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Right;
        KRef_SetObjectPath(field, x);
    }
}

```

```

KINLINE CMPIStatus GadgetRef_Set_Right(
    GadgetRef* self,
    const WidgetRef* x)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Right;
        return KRef_Set(field, &x->__base);
    }
    CMReturn(CMPI_RC_ERR_FAILED);
}

```

```

KINLINE void GadgetRef_Null_Right(
    GadgetRef* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Right;
        KRef_Null(field);
    }
}

```

```

KINLINE void GadgetRef_Clr_Right(
    GadgetRef* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Right;
        KRef_Clr(field);
    }
}

```

```

/*

```

```

*****
```

```

**

```

```

** struct Gadget

```

```

**
**=====
*/

/* classname=KC_Gadget */
typedef struct _Gadget
{
    KBase __base;
    /* KC_Gadget features */
    const KRef Left; /* Widget */
    const KRef Right; /* Widget */
}
Gadget;

static const unsigned char __Gadget_sig[] =
{
    0x09,0x4b,0x43,0x5f,0x47,0x61,0x64,0x67,0x65,0x74,0x00,0x02,0x4e,0x04,0x4c,
    0x65,0x66,0x74,0x00,0x4e,0x05,0x52,0x69,0x67,0x68,0x74,0x00,
};

KINLINE void Gadget_Init(
    Gadget* self,
    const CMPIBroker* cb,
    const char* ns)
{
    const unsigned char* sig = __Gadget_sig;
    KBase_Init(&self->__base, cb, sizeof(*self), sig, ns);
    ((KRef*)&self->Left)->__sig = __Widget_sig;
    ((KRef*)&self->Right)->__sig = __Widget_sig;
}

KINLINE CMPIStatus Gadget_InitFromInstance(
    Gadget* self,
    const CMPIBroker* cb,
    const CMPIInstance* x)
{
    Gadget_Init(self, cb, NULL);
    return KBase_FromInstance(&self->__base, x);
}

KINLINE CMPIStatus Gadget_InitFromObjectPath(
    Gadget* self,
    const CMPIBroker* cb,
    const CMPIObjectPath* x)
{
    Gadget_Init(self, cb, NULL);
    return KBase_FromObjectPath(&self->__base, x);
}

```

```

}

KINLINE void Gadget_Print(
    const Gadget* self,
    FILE* os)
{
    KBase_Print(os, &self->__base, 'i');
}

KINLINE CMPIInstance* Gadget_ToInstance(
    const Gadget* self,
    CMPIStatus* status)
{
    return KBase_ToInstance(&self->__base, status);
}

KINLINE CMPIObjectPath* Gadget_ToObjectPath(
    const Gadget* self,
    CMPIStatus* status)
{
    return KBase_ToObjectPath(&self->__base, status);
}

KINLINE const char* Gadget_NameSpace(
    Gadget* self)
{
    if (self && self->__base.magic == KMAGIC)
        return self->__base.ns ? KChars(self->__base.ns) : NULL;
    return NULL;
}

KINLINE void Gadget_SetObjectPath_Left(
    Gadget* self,
    const CMPIObjectPath* x)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Left;
        KRef_SetObjectPath(field, x);
    }
}

KINLINE CMPIStatus Gadget_Set_Left(
    Gadget* self,
    const WidgetRef* x)
{
    if (self && self->__base.magic == KMAGIC)

```

```

    {
        KRef* field = (KRef*)&self->Left;
        return KRef_Set(field, &x->__base);
    }
    CMReturn(CMPI_RC_ERR_FAILED);
}

KINLINE void Gadget_Null_Left(
    Gadget* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Left;
        KRef_Null(field);
    }
}

KINLINE void Gadget_Clr_Left(
    Gadget* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Left;
        KRef_Clr(field);
    }
}

KINLINE void Gadget_SetObjectPath_Right(
    Gadget* self,
    const CMPIObjectPath* x)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Right;
        KRef_SetObjectPath(field, x);
    }
}

KINLINE CMPIStatus Gadget_Set_Right(
    Gadget* self,
    const WidgetRef* x)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Right;
        return KRef_Set(field, &x->__base);
    }
}

```

```

    CMReturn(CMPI_RC_ERR_FAILED);
}

KINLINE void Gadget_Null_Right(
    Gadget* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Right;
        KRef_Null(field);
    }
}

KINLINE void Gadget_Clr_Right(
    Gadget* self)
{
    if (self && self->__base.magic == KMAGIC)
    {
        KRef* field = (KRef*)&self->Right;
        KRef_Clr(field);
    }
}

/*
**=====
**
** Gadget methods
**
**=====
*/

KINLINE CMPIStatus Gadget_InvokeMethod(
    const CMPIBroker* cb,
    CMPIMethodMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const char* meth,
    const CMPIArgs* in,
    CMPIArgs* out)
{
    GadgetRef self;

    KReturnIf(GadgetRef_InitFromObjectPath(&self, cb, cop));

    KReturn(ERR_METHOD_NOT_FOUND);
}

```

```
}
```

```
#endif /* _konkrete_Gadget_h */
```

Appendix D GadgetProvider.c Listing

```
#include <konkret/konkret.h>
#include "Gadget.h"

static const CMPIBroker* _broker;

static void GadgetInitialize()
{
}

CMPIStatus GadgetCleanup(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    CMPIBoolean term)
{
    CMReturn(CMPI_RC_OK);
}

CMPIStatus GadgetEnumInstanceNames(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop)
{
    return KDefaultEnumerateInstanceNames(
        _broker, mi, cc, cr, cop);
}

CMPIStatus GadgetEnumInstances(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const char** properties)
{
    CMReturn(CMPI_RC_OK);
}

CMPIStatus GadgetGetInstance(
    CMPIInstanceMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const char** properties)
{
    return KDefaultGetInstance(
```

```
        _broker, mi, cc, cr, cop, properties);  
}
```

```
CMPIStatus GadgetCreateInstance(  
    CMPIInstanceMI* mi,  
    const CMPIContext* cc,  
    const CMPIResult* cr,  
    const CMPIObjectPath* cop,  
    const CMPIInstance* ci)  
{  
    CMReturn(CMPI_RC_ERR_NOT_SUPPORTED);  
}
```

```
CMPIStatus GadgetModifyInstance(  
    CMPIInstanceMI* mi,  
    const CMPIContext* cc,  
    const CMPIResult* cr,  
    const CMPIObjectPath* cop,  
    const CMPIInstance* ci,  
    const char**properties)  
{  
    CMReturn(CMPI_RC_ERR_NOT_SUPPORTED);  
}
```

```
CMPIStatus GadgetDeleteInstance(  
    CMPIInstanceMI* mi,  
    const CMPIContext* cc,  
    const CMPIResult* cr,  
    const CMPIObjectPath* cop)  
{  
    CMReturn(CMPI_RC_ERR_NOT_SUPPORTED);  
}
```

```
CMPIStatus GadgetExecQuery(  
    CMPIInstanceMI* mi,  
    const CMPIContext* cc,  
    const CMPIResult* cr,  
    const CMPIObjectPath* cop,  
    const char* lang,  
    const char* query)  
{  
    CMReturn(CMPI_RC_ERR_NOT_SUPPORTED);  
}
```

```
CMPIStatus GadgetAssociationCleanup(  
    CMPIAssociationMI* mi,  
    const CMPIContext* cc,
```

```
    CMPIBoolean term)
{
    CMReturn(CMPI_RC_OK);
}

CMPIStatus GadgetAssociators(
    CMPIAssociationMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const char* assocClass,
    const char* resultClass,
    const char* role,
    const char* resultRole,
    const char** properties)
{
    if (!assocClass)
        assocClass = "KC_Gadget";

    return KDefaultAssociators(_broker, mi, cc, cr, cop, assocClass,
        resultClass, role, resultRole, properties);
}

CMPIStatus GadgetAssociatorNames(
    CMPIAssociationMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const char* assocClass,
    const char* resultClass,
    const char* role,
    const char* resultRole)
{
    if (!assocClass)
        assocClass = "KC_Gadget";

    return KDefaultAssociatorNames(_broker, mi, cc, cr, cop,
        assocClass, resultClass, role, resultRole);
}

CMPIStatus GadgetReferences(
    CMPIAssociationMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const char* assocClass,
    const char* role,
```

```
    const char** properties)
{
    if (!assocClass)
        assocClass = "KC_Gadget";

    return KDefaultReferences(_broker, mi, cc, cr, cop, assocClass,
        role, properties);
}
```

```
CMPIStatus GadgetReferenceNames(
    CMPIAssociationMI* mi,
    const CMPIContext* cc,
    const CMPIResult* cr,
    const CMPIObjectPath* cop,
    const char* assocClass,
    const char* role)
{
    if (!assocClass)
        assocClass = "KC_Gadget";

    return KDefaultReferenceNames(
        _broker, mi, cc, cr, cop, assocClass, role);
}
```

```
CMInstanceMIStub(
    Gadget,
    Gadget,
    _broker,
    GadgetInitialize())
```

```
CMAssociationMIStub(
    Gadget,
    Gadget,
    _broker,
    GadgetInitialize())
```