EUROPEAN SPACE AGENCY
DIRECTORATE OF OPERATIONS AND INFRASTRUCTURE

SMP 2.0 C++ Model Development Kit

EGOS-SIM-GEN-TN-1001

Issue 1 Revision 2

28 October 2005

This Page is Intentionally left Blank

ABSTRACT

This technical note contains the documentation of the model development kit, which supports implementing models for the SMP2 standard.

DOCUMENT APPROVAL

Prepared by	Organisation	Signature	Date
Peter Fritzen Peter Ellsiepen	VEGA VEGA		28 October 2005

Verified by	Organisation	Signature	Date
Christine Dingeldey	VEGA		28 October 2005

Approved by	Organisation	Signature	Date
Niklas Lindman	ESOC/OPS-GIC		

DOCUMENT STATUS SHEET

1. Issue	2. Revision	3. Date	4. Reason for Change
1	**0**	**28 February 2005**	First Issue of this document.
1	**2**	**28 October 2005**	First Update of this document.

Note: To align issue and revision of this document with the SMP2 Standard, the first update has been called Issue 1 Revision 2 rather than Issue 1 Revision 1.

This Page is Intentionally left Blank

DOCUMENT CHANGE RECORD

DOCUMENT CHANGE RECORD Changes from SMP 2.0 C++ Model Development Kit Issue 1 to SMP 2.0 C++ Model Development Kit Issue 1 Revision 2			**DCI NO**	N/A
			DATE	28 October 2005
			ORIGINATOR	SMP CCB
			APPROVED BY	Niklas Lindman
1. PAGE	**2. PARAGRAPH**	**3. ISSUE**	**4. CHANGES MADE**	
15	1.4	46	SMP Handbook and Alpha Specification moved from Applicable Documents to Reference Documents.	
19	2	16	`assert` removed from example code.	
25	2.3	-4	`BUFFER_SIZE` constant added.	
25	2.3.3		Compare operators added.	
26	2.3.7		Empty `Uuid` added.	
29	2.5	54	`String` template added.	
30	2.6	54	`Array` template added.	
34	3.1.2.2		Section updated for updates in `IModel` interface.	
41	3.2.3.1	-2	`VoidEventSink` template added.	
43	3.2.3.2	-2	`VoidEventSource` template added.	
47	3.3.1.1	10	Note added that owner of `EntryPoint` needs to implement `IComponent`.	
60	3.5.1.1		`Specification` and `Implementation` Uuids added.	
61	4.1	23	Access to services replaced by new convenience methods.	
67	4.2		`IModel` implementation added for model states.	
73	4.3		`InvalidModelState` exception added.	
76	4.4		`IModel` implementation added for model states.	
87	4.5		`InvalidModelState` exception added.	

This Page is Intentionally left Blank

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

1. INTRODUCTION

This document introduces the C++ Model Development Kit (**MDK**) that has been developed in order to support development of C++ models in compliance with the Simulation Model Portability (**SMP**) 2 standard. For each interface of the SMP2 Component Model, instructions are provided on how it can be implemented by making use of the C++ MDK.

1.1 Purpose

The purpose of this document is to give support to those that want to implement SMP2 compliant C++ models based on the C++ MDK. While the MDK itself is not part of the standard, it eases the migration to it by providing an implementation of almost every interface a model may implement.

The MDK itself is delivered as a set of C++ files together with a Makefile. In addition, some example models are provided. Documentation can be generated using doxygen, but this only describes each function individually, but not within its context. This document tries to add an overview, and guides the reader on how to implement a specific model feature.

It is recommended to first read the SMP 2.0 Component Model document, and then the SMP 2.0 C++ Mapping, before using the MDK to develop C++ models. This document assumes that the reader is familiar with the SMP 2.0 Component Model and its mapping to the C++ language.

1.2 Scope

In the first place, this document targets the model developer. It provides examples and instructions on how to efficiently code SMP2 models in C++. This is especially useful for managed models: Without the MDK, it takes some effort to add the management interfaces to an unmanaged model. Using the MDK, providing managed models becomes almost as easy as providing unmanaged models. This should encourage people to develop managed models from the early start, even if they don't immediately see a need for it.

In the second place, this document targets tool developers, namely people that want to write a Code Generator for SMP2 models. While not mandatory, basing a code generator on the MDK base classes reduces the amount of auto-generated code, and increases flexibility. Should one of the base implementations have a problem, fixing this problem in the MDK will automatically affect all models based on it, while manual code in each individual model has to be corrected in each model.

1.3 Definitions, acronyms and abbreviations

AD	Applicable Document
DLL	Dynamic Link Library
DSO	Dynamic Shared Object
ESOC	European Space Operations Centre
MDK	Model Development Kit
N/A	Not Applicable
RD	Reference Document
SMP	Simulation Model Portability
TBC	To be confirmed
TBD	To be defined

1.4 References

1.4.1 Applicable Documents

Applicable documents are denoted with AD-n where n is the number in the following list:

AD-1 SMP 2.0 Handbook
EGOS-SIM-GEN-TN-0099, Issue 1.2, 28-Oct-2005

AD-2 SMP 2.0 Metamodel
EGOS-SIM-GEN-TN-0100, Issue 1.2, 28-Oct-2005

AD-3 SMP 2.0 Component Model
EGOS-SIM-GEN-TN-0101, Issue 1.2, 28-Oct-2005

AD-4 SMP 2.0 C++ Mapping
EGOS-SIM-GEN-TN-0102, Issue 1.2, 28-Oct-2005

1.4.2 Reference Documents

Reference documents are denoted with RD-n where n is the number in the following list:

RD-1 Simulation Model Portability Handbook
EWP-2080, Issue 1.1, 31-Oct-2000

RD-2 SMP2 Alpha Specification
SIM-GST-TN-0045-TOS-GIC, Issue 1.0, 30-Dec-2003

1.5 Overview

This document introduces the classes and templates of the SMP 2.0 C++ Model Development Kit. As the MDK is closely related to the (C++ mapping of the) Component Model, the structure of this document reflects the structure of the SMP 2.0 Component Model document.

1.5.1 Objects

In order to allow harmonisation of components with other SMP2 elements, SMP2 first defines what an object is (`IObject`). Although most elements of the SMP2 component model are components, these components expose some objects, which are derived directly from the `IObject` interface. These objects include entry points, event sources and event sinks, and containers and references.

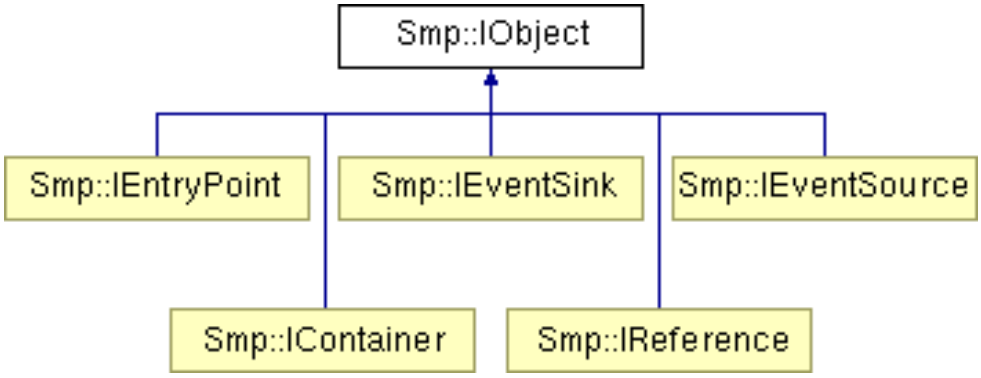

Figure 1-1: Object Types

The MDK provides a base implementation of the `IObject` interface in its `Object` class. See section 3.1.1 for details on objects.

1.5.2 Components

To allow treating all components in a similar way, SMP2 defines what a component is (i.e. the `IComponent` interface). The three most important interfaces derived from this base interface are `IModel` for models, `IService` for services, and `ISimulator` for the simulation environment (where `ISimulator` is not derived immediately from `IComponent`, but from `IComposite`).

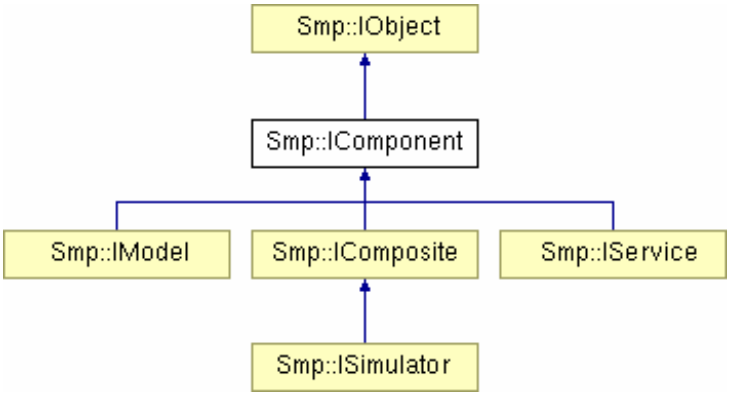

Figure 1-2: Component Types

For the MDK, only the `IComponent` and `IModel` interface are of interest, which are implemented by the `Component` and `Model` base classes. See section 3.1 for details on Objects and Components.

1.5.3 Component Mechanisms

Further, SMP2 defines standard mechanisms how to enhance components, for example for component aggregation (`IAggregate`), component composition (`IComposite`), inter-component events (`IEventProvider` and `IEventConsumer`), dynamic invocation (`IDynamicInvocation`) and self-persistence (`IPersist`).

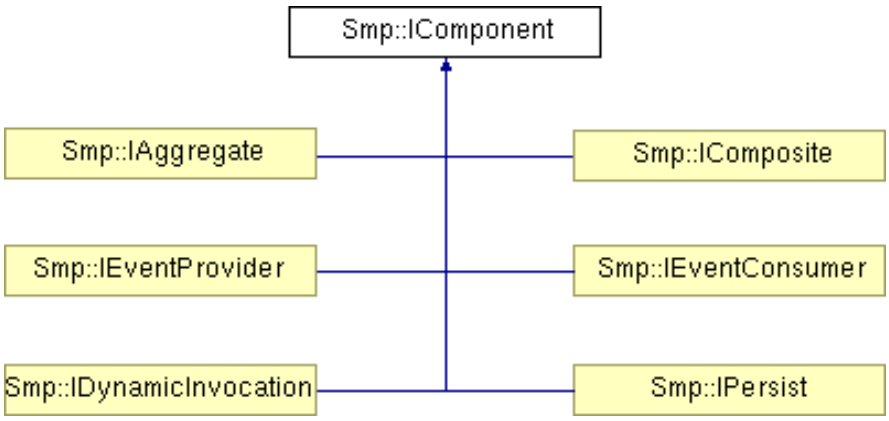

Figure 1-3: Component Mechanisms

The MDK provides classes implementing aggregation (`Aggregate`), composition (`Composite`), and inter-component events (`EventProvider` and `EventConsumer`). The `Model` base class implements two of the three methods of `IDynamicInvocation`, but persistence always has to be implemented by the model itself (the purpose of `IPersist` is to provide custom persistence). See section 3.2 for details on Component Mechanisms.

1.5.4 Model Mechanisms

The most important component type of SMP2 is the Model. While implementing the mandatory `IModel` interface is sufficient to make a model SMP2 compliant, this does not allow using most of the SMP2 mechanisms. Therefore, optional extensions of the `IModel` interface do exist providing optional features for models that go beyond those for components. A fundamental one is that of an entry point publisher (`IEntryPointPublisher`), another optional feature is the implementation of `IManagedModel`.

The MDK provides an implementation of `IEntryPointPublisher` in the `EntryPointPublisher` class, and an implementation of `IManagedModel` in the `ManagedModel` class. See section 3.3 for details on Model Mechanisms, and section 3.4.3 for Managed Model Mechanisms.

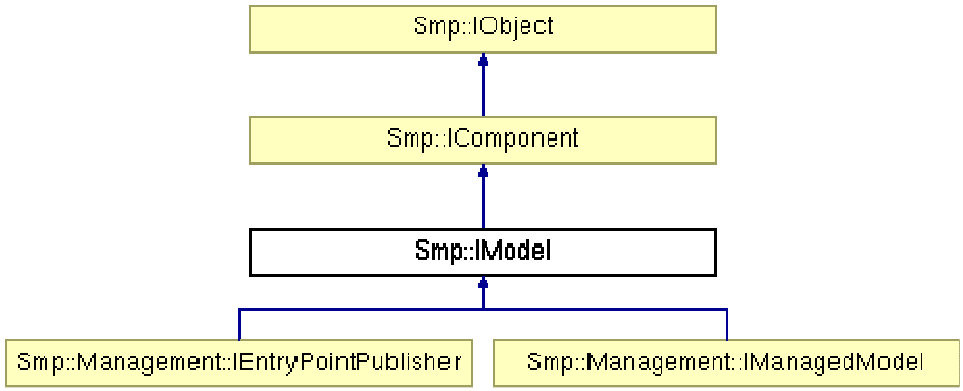

Figure 1-4: Model Mechanisms

1.5.5 Management Interfaces

To support dynamically configured simulations, where all information about the models, their links and initial values is read from an external source (typically XML files), SMP2 defines optional mechanisms for managed components. These allow setting their properties (name, description, and parent) from external applications, and provide mechanisms to query their elements (field values, entry points, event sources and sinks) by name.

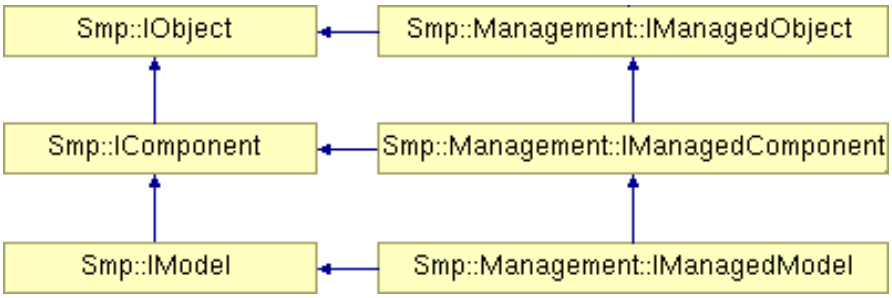

Figure 1-5: Management Interfaces

The MDK provides an implementation for each of the managed interfaces. See section 3.4 for details on Management Interfaces.

1.5.6 Simulation Environments

The interfaces that a Simulation Environment has to implement are out of scope for the MDK, as they are only an input into the models, but not actively implemented by them.

1.5.7 Simulation Services

Simulation Services are not implemented by models, but used by them. Therefore, the MDK does not provide support for implementing these interfaces.

1.5.8 Mandatory and Optional Interfaces

The MDK provides an implementation of almost all interfaces, may they be mandatory or optional.

1.5.9 Exceptions

The mapping of the SMP2 exceptions to C++ is provided as part of the C++ Mapping, and is not touched by the MDK.

2. SIMPLE TYPES

This section introduces wrappers for the simple types Duration, DateTime, Uuid and AnySimple.

2.1 Duration Wrapper

```
namespace Smp
{
    namespace Mdk
    {
        /// Duration wrapper for Smp::Duration type.
        struct Duration
        {
            Smp::Duration                ticks;    // 64 bits
            static const Smp::Duration   TicksPerSecond = 1000*1000*1000;

            // Constructors
            Duration();
            Duration(const Smp::Duration& source);
            Duration(const Smp::Int32 days,
                     const Smp::Int16 hours,
                     const Smp::Int16 minutes,
                     const Smp::Int16 seconds,
                     const Smp::Int32 nanoseconds = 0);
            Duration(const Smp::Float64 seconds);
            Duration(const char* string);

            // Assignment Operators
            Smp::Mdk::Duration& operator = (const Smp::Duration& source);
            Smp::Mdk::Duration& operator = (const Smp::Mdk::Duration& source);
            Smp::Mdk::Duration& operator = (const Smp::Float64 seconds);
            Smp::Mdk::Duration& operator = (const Smp::String8 source);

            /// Math Operators
            Smp::Mdk::Duration operator - (void) const;
            Smp::Mdk::Duration operator + (Smp::Mdk::Duration duration) const;
            Smp::Mdk::Duration operator - (Smp::Mdk::Duration duration) const;
            void operator += (Smp::Mdk::Duration duration);
            void operator -= (Smp::Mdk::Duration duration);

            /// Compare Operators
            bool operator >  (Smp::Mdk::Duration duration) const;
            bool operator <  (Smp::Mdk::Duration duration) const;
            bool operator == (Smp::Mdk::Duration duration) const;
            bool operator != (Smp::Mdk::Duration duration) const;

            // Setter & Getter
            bool Set(const Smp::Float64 seconds);
            bool Set(const char* string);
            bool Set(const Smp::Int32 days,
                     const Smp::Int16 hours,
                     const Smp::Int16 minutes,
                     const Smp::Int16 seconds,
                     const Smp::Int32 nanoseconds = 0);
            void Get(char* string) const;
            void Get(Smp::Int32& days,
                     Smp::Int16& hours,
                     Smp::Int16& minutes,
                     Smp::Int16& seconds,
                     Smp::Int32& nanoseconds) const;
            Smp::Float64 Get() const;
        };
    }
}
```

The Smp Duration type measures time in nanoseconds. In C++, it is mapped to Int64. While this is an accurate measurement of time, it is not convenient to e.g. enter a duration in seconds, from time elements (days/hours/minutes/seconds/nanoseconds), or even from a string representation. Therefore, the Mdk provides a wrapper with this extra functionality.

2.1.1 Constructors

The Duration structure provides different constructors. A duration can be created from a double in seconds, from a string using a duration format, from time elements (days, hours, minutes, seconds, nanoseconds), and from another duration.

```
// Create a Duration
d1 = Duration(0.5);              //    00:00:00.5
d2 = Duration("1.02:15:00.501");//  1.02:15:00.501
d3 = Duration(-1,-2,-15,0,0);    // -1.02:15:00
d4 = Duration(d2);               //  1.02:15:00.501
```

As a duration can be positive as well as negative, each constructor can be called with a positive or a negative value. For the string constructor, a negative sign can be used, while for the constructor with time elements, all elements need to be specified negative.

2.1.2 Assignment Operators

Alternatively, one of the assignment operators can be used.

```
// Assign a Duration
d1 = 0.5;                        //    00:00:00.5
d2 = "1.02:15:00.501";           //  1.02:15:00.501
d3 = "-1.02:15:00";              // -1.02:15:00
d4 = d2;                         //  1.02:15:00.501
```

2.1.3 Math Operators

The basic operators have been overloaded to allow modifying durations.

```
// Modify a Duration
d5 = d3 + d4;                    //    00:00:00.501
d5 = d5 - d1;                    //    00:00:00.001
d4 += d3;                        //    00:00:00.501
d4 -= d1;                        //    00:00:00.001
d5 = -d4;                        //   -00:00:00.001
```

2.1.4 Setter and Getter methods

To set the value of a duration, the overloaded `Set()` methods can be used.

```
// Set a Duration
d1.Set(-0.5);              //   -00:00:00.5
d2.Set("-1.02:15:00.501"); // -1.02:15:00.501
d3.Set(  1, 2,15,0, 0);    //  1.02:15:00
```

To get the value of a duration, the overloaded `Get()` methods can be used.

```
char durationString[32];
Smp::Int32 dd, ns;
Smp::Int16 hh, mm, ss;
// Get a Duration
d4.Get(durationString);          // Get string representation
d4.Get(dd, hh, mm, ss, ns);      // Get time elements
Smp::Float64 secs = d4.Get();    // Get seconds
```

2.1.5 Convert to Smp Duration

When an `Smp::Duration` value has to be passed to a method of an Smp interface, this can be extracted from the `Smp::Mdk::Duration` as the number of `ticks`.

```
Smp::Duration duration = d5.ticks;
```

2.1.6 Duration Format

The Mdk Duration wrapper class provides methods to convert a duration value to and from a string. These methods use a format based on the string format for time, but amended by additional elements for full days, additional nanoseconds, and an optional sign. The format is defined as follows:

```
durationFormat = [+|-][days.]hh:mm:ss[.nanoseconds]
```

The following holds:

- Every duration needs to specify at least hours, minutes and seconds, separated by a colon (":").
- Fractions of seconds can be added at the end using a dot (".") as separator.
- Full days can be added at the beginning using a dot (".") as separator.
- The duration may start with a positive ("+") or negative ("-") sign.

Examples:

```
// Duration format
d1.Set(         "00:00:00");
d1.Set(    "123.00:00:00");
d1.Set(         "00:00:00.123456789");
d1.Set(    "123.00:00:00.123456789");
d1.Set("-100000.00:00:00.123456789");
```

When using the overloaded `Get()` method, a character buffer has to be passed that has to be large enough to hold the duration representation. As a duration can have more than 100.000 days[1] (6 digits), and up to 9 digits can be used for the additional nanoseconds, the whole string can take up to 26 characters (plus an extra null termination character). Hence, it is strongly recommended to always provide a buffer of at least 27 characters in size.

[1] *A duration uses a signed 64 bit integer value measured in nanoseconds. This allows storing values of almost 106752 days in a duration. This translates to more than 292 years.*

2.2 DateTime Wrapper

```
namespace Smp
{
    namespace Mdk
    {
        /// DateTime wrapper for Smp::DateTime type.
        struct DateTime
        {
            Smp::DateTime           ticks;      // 64 bits
            static const Smp::UInt8 DaysInMonth[12];
            static const Smp::Int64 TicksPerSecond  = 1000*1000*1000;
            static const Smp::Int64 TicksPerDay     = 24*60*60*TicksPerSecond;
            static const Smp::Int64 TicksPerCentury = 36524*TicksPerDay;

            // Constructors
            DateTime();
            DateTime(const char* string);
            DateTime(const Smp::DateTime& source);
            DateTime(const Smp::Float64 days);
            DateTime(const Smp::Int16 year,
                     const Smp::Int16 month,
                     const Smp::Int16 day,
                     const Smp::Int16 hours,
                     const Smp::Int16 minutes,
                     const Smp::Int16 seconds,
                     const Smp::Int32 nanoseconds = 0);

            // Assignment Operators
            Smp::Mdk::DateTime& operator = (const Smp::DateTime& source);
            Smp::Mdk::DateTime& operator = (const Smp::Mdk::DateTime& source);
            Smp::Mdk::DateTime& operator = (const Smp::Float64 days);

            // Math Operators
            Smp::Mdk::Duration operator - (Smp::Mdk::DateTime dateTime) const;
            Smp::Mdk::DateTime operator + (Smp::Mdk::Duration duration) const;
            Smp::Mdk::DateTime operator - (Smp::Mdk::Duration duration) const;
            void operator += (Smp::Mdk::Duration duration);
            void operator -= (Smp::Mdk::Duration duration);

            // Compare Operators
            bool operator >  (Smp::Mdk::DateTime dateTime) const;
            bool operator <  (Smp::Mdk::DateTime dateTime) const;
            bool operator == (Smp::Mdk::DateTime dateTime) const;
            bool operator != (Smp::Mdk::DateTime dateTime) const;

            // Setter & Getter
            bool Set(const char* string);
            bool Set(const Smp::Float64 days);
            bool Set(const Smp::Int16 year,
                     const Smp::Int16 month,
                     const Smp::Int16 day,
                     const Smp::Int16 hours,
                     const Smp::Int16 minutes,
                     const Smp::Int16 seconds,
                     const Smp::Int32 nanoseconds = 0);
            void Get(char* string) const;
            void Get(Smp::Int16& year,
                     Smp::Int16& month,
                     Smp::Int16& day,
                     Smp::Int16& hours,
                     Smp::Int16& minutes,
                     Smp::Int16& seconds,
                     Smp::Int32& nanoseconds) const;
            Smp::Float64 Get() const;

            // Static Functions
            static bool IsLeapYear(int year);
        };
    }
}
```

The Smp DateTime type measures time in nanoseconds. In C++, it is mapped to Int64. While this is an accurate measurement of time, it is not convenient to e.g. enter a date in days, from date and time elements (day/month/year, hours/minutes/seconds/nanoseconds), or even from a string representation. Therefore, the Mdk provides a wrapper with this extra functionality.

2.2.1 Constructors

The DateTime structure provides different constructors. A date can be created from a double in days, from a string using a date/time format, from date and time elements (day, month, year, hours, minutes, seconds, nanoseconds), and from another date.

```
// Create a DateTime
dt1 = DateTime();                       // 01.01.2000 12:00:00
dt2 = DateTime(0.5);                    // 02.01.2000 00:00:00
dt3 = DateTime("28.02.2005 15:20:30");  // 28.02.2005 15:20:30
dt4 = DateTime(2005, 2, 28,15,20,30);   // 28.02.2005 15:20:30
dt5 = DateTime(dt2);                    // 02.01.2000 00:00:00
```

2.2.2 Assignment Operators

Alternatively, one of the assignment operators can be used.

```
// Assign a DateTime
dt1 = 0.5;                              // 02.01.2000 00:00:00
dt2 = "28.02.2005 15:20:30";            // 28.02.2005 15:20:30
dt3 = "01.01.1900";                     // 01.01.1900
dt4 = dt2;                              // 28.02.2005 15:20:30
```

2.2.3 Math Operators

The basic operators have been overloaded to allow modifying dates. As the difference between two dates is a duration, these operators involve the Mdk Duration type as well.

```
Duration du1;

// Modify a DateTime
dt5 = dt3 + Duration("365.00:00:00");   // 01.01.1901
du1 = dt5 - dt3;                        //        365.00:00:00
dt5 -= du1;                             // 01.01.1900
```

The difference of two DateTime values is a Duration, while the sum of dates is not defined. Similarly, a Duration can be added to and subtracted from a DateTime, which results in another DateTime.

As a date can never be negative, there is no unary negative operator.

2.2.4 Setter and Getter methods

To set the value of a date, the overloaded `Set()` methods can be used.

```
// Set a DateTime
dt1.Set(-0.5);                          // 01.01.2000 00:00:00
dt2.Set("01.01.2000");                  // 01.01.2000 00:00:00
dt3.Set(2005, 2, 28, 15, 20, 30);       // 28.02.2005 15:20:30
```

To get the value of a duration, the overloaded `Get()` methods can be used.

```
Smp::Int32 ns;
Smp::Int16 day, month, year, hour, min, sec;
Smp::Float64 days;

// Get a DateTime
days = dt3.Get();                       // Get days
dt5.Get(dateTimeString);                // Get string representation
dt5.Get(year, month, day,               // Get date and time elements
        hour, min, sec, ns);
```

2.2.5 Convert to Smp DateTime

When an `Smp::DateTime` value has to be passed to a method of an Smp interface, this can be extracted from the `Smp::Mdk::DateTime` as the number of `ticks`.

```
Smp::DateTime DateTime = dt5.ticks;
```

2.2.6 Date and Time Format

The Mdk DateTime wrapper class provides methods to convert a date value to and from a string. These methods use a format based on the string formats for date and time, but amended by an additional element for nanoseconds. The format is defined as follows:

```
dateTimeFormat = day.month.year[ hh:mm:ss[.nanoseconds]]
```

The following holds:

- Every date needs to specify at least day, month and year, separated by a dot (“.”).
- A time of the day can be added separated by a space (“ “), which consists of hour, minute and second separated by a colon (“:”).
- Fractions of seconds can be added at the end of the time using a dot (“.”) as separator.

Examples:

```
// DateTime format
dt1.Set("01.01.2000");
dt2.Set("01.01.2000 00:00:00");
dt3.Set("01.01.2000 00:00:00.123456789");
```

When using the overloaded `Get()` method, a character buffer has to be passed that has to be large enough to hold the date and time representation. As a date has 10 digits, a time has 8 digits, and up to 9 digits can be used for the additional nanoseconds, the whole string can take up to 29 characters (plus an extra null termination character). Hence, it is strongly recommended to always provide a buffer of at least 30 characters in size.

2.3 Uuid Wrapper

```
namespace Smp
{
    namespace Mdk
    {
        /// Uuid wrapper for Smp::Uuid type.
        struct Uuid : public Smp::Uuid
        {
            // Minimum size of a buffer that can store a Uuid as string.
            static const BUFFER_SIZE = 37;

            // Constructors
            Uuid();
            Uuid(const Smp::Uuid& source);
            Uuid(const char* string);

            // Operators
            Uuid& operator = (const Smp::Uuid& source);
            Uuid& operator = (const char* string);
            bool operator == (const Smp::Uuid uuid) const;
            bool operator != (const Smp::Uuid uuid) const;

            // Setter & Getter
            bool Set(const char* string);
            void Get(char* string) const;
        };
    }
}
```

The Smp Uuid type is a structure with the fields that identify a universally unique identifier. In C++, it is mapped to a `struct`. While it is possible to define a Uuid via its individual fields, it is not convenient to do so. Typically, it is more convenient to enter and show a Uuid using a string representation. Therefore, the Mdk provides a wrapper with this extra functionality.

2.3.1 Constructors

The Mdk Uuid structure provides different constructors. A Uuid can be created from a string representation, and from another Uuid.

```
    // Create a Uuid
    uuid1 = Uuid();
    uuid2 = Uuid("12345678-1234-1234-1234-123456789ABC");
    uuid3 = Uuid(uuid1);
```

2.3.2 Assignment Operators

Alternatively, one of the assignment operators can be used.

```
    // Assign a Uuid
    uuid2 = "12345678-1234-1234-1234-123456789ABC";
    uuid3 = uuid1;
```

2.3.3 Compare Operators

Additionally, operators to compare Uuids with each other exist.

```
    // Compare two Uuids
    if (uuid2 == uuid3) {}
    if (uuid3 != uuid1) {}
```

2.3.4 Setter and Getter methods

To set the value of a Uuid, the `Set()` method can be used.

```
// Set a Uuid
uuid2.Set("12345678-1234-1234-1234-123456789ABC");
```

To get the value of a Uuid, the `Get()` method can be used.

```
// Get a Uuid
uuid2.Get(uuidString);
```

2.3.5 Convert to Smp Uuid

As `Smp::Mdk::Uuid` inherits from `Smp::Uuid`, an Mdk Uuid can be used whenever an Smp Uuid is expected.

2.3.6 Uuid Format

The Mdk Uuid wrapper class provides methods to convert a Uuid value to and from a string. These methods use a format based on the standard string format. The format is defined as follows:

```
uuidFormat = XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
```

Every string always has to specify all fields. However, constructor and `Set()` method are flexible and ignore any characters that are not part of a hexadecimal number.

Examples:

```
// Uuid format
uuid1.Set("12345678123412341234123456789ABC");
uuid1.Set("12345678-1234-1234-1234-123456789ABC");
uuid1.Set("{12345678-1234-1234-1234-123456789ABC}");
```

When using the `Get()` method, a character buffer has to be passed that has to be large enough to hold the Uuid representation. As a Uuid has 128 bits, which are represented using 32 hexadecimal digits, the whole string is 36 characters long (plus an extra null termination character). Hence, it is strongly recommended to always provide a buffer of at least 37 characters in size.

2.3.7 Empty Uuid

In some cases (e.g. in the type registry), it is required to use a Uuid that represents “no type” (e.g. to say that a class has no base class). For this purpose, a constant has been defined in the Mdk.

```
/// Uuid that does not specify a type, but stands for "no type".
static const Uuid NullUuid = Uuid("00000000-0000-0000-0000-000000000000");
```

2.4 AnySimple Wrapper

```
namespace Smp
{
    namespace Mdk
    {
        /// AnySimple wrapper for Smp::AnySimple type.
        struct AnySimple : public Smp::AnySimple
        {
            // Operators
            AnySimple& operator = (const Smp::AnySimple& source);

            // Setter methods
            void Set(const Smp::Int64 v, Smp::SimpleTypeKind t) throw
                (Smp::InvalidAnyType);

            void Set(const Smp::Char8   v);
            void Set(const Smp::Bool    v);
            void Set(const Smp::Int8    v);
            void Set(const Smp::Int16   v);
            void Set(const Smp::Int32   v);
            void Set(const Smp::Int64   v);
            void Set(const Smp::UInt8   v);
            void Set(const Smp::UInt16  v);
            void Set(const Smp::UInt32  v);
            void Set(const Smp::UInt64  v);
            void Set(const Smp::Float32 v);
            void Set(const Smp::Float64 v);
            void Set(const Smp::Mdk::DateTime& v);
            void Set(const Smp::Mdk::Duration& v);
            void Set(const Smp::AnySimple& source) throw (Smp::InvalidAnyType);

            // Getter methods
            void Get(Smp::Char8&   v) throw (Smp::InvalidAnyType);
            void Get(Smp::Bool&    v) throw (Smp::InvalidAnyType);
            void Get(Smp::Int8&    v) throw (Smp::InvalidAnyType);
            void Get(Smp::Int16&   v) throw (Smp::InvalidAnyType);
            void Get(Smp::Int32&   v) throw (Smp::InvalidAnyType);
            void Get(Smp::Int64&   v) throw (Smp::InvalidAnyType);
            void Get(Smp::UInt8&   v) throw (Smp::InvalidAnyType);
            void Get(Smp::UInt16&  v) throw (Smp::InvalidAnyType);
            void Get(Smp::UInt32&  v) throw (Smp::InvalidAnyType);
            void Get(Smp::UInt64&  v) throw (Smp::InvalidAnyType);
            void Get(Smp::Float32& v) throw (Smp::InvalidAnyType);
            void Get(Smp::Float64& v) throw (Smp::InvalidAnyType);
        };
    }
}
```

The Smp AnySimple type is a discriminated structure with type and value, but it does not allow assigning type and value in one operation. Therefore, the Mdk provides a wrapper with this extra functionality.

2.4.1 Assignment Operators

The assignment operator can be used to assign type and value of another `AnySimple`, which can be either an `Smp::AnySimple` or another `Smp::Mdk::AnySimple`.

```
    // Assign type and value
    mdkAny = smpAny;
```

2.4.2 Setter methods

To set type and value of an `AnySimple`, the overloaded `Set()` methods can be used. These assign a type based on the type of the value argument.

An overloaded method exists for almost every primitive type, except for `DateTime` and `Duration`, which are mapped to `Int64`.

```
// Set AnySimple
mdkAny.Set('x');                    assert(mdkAny.type == Smp::ST_Char8);
mdkAny.Set(false);                  assert(mdkAny.type == Smp::ST_Bool);
mdkAny.Set((Smp::Int8)    0);       assert(mdkAny.type == Smp::ST_Int8);
mdkAny.Set((Smp::Int16)   0);       assert(mdkAny.type == Smp::ST_Int16);
mdkAny.Set((Smp::Int32)   0);       assert(mdkAny.type == Smp::ST_Int32);
mdkAny.Set((Smp::Int64)   0);       assert(mdkAny.type == Smp::ST_Int64);
mdkAny.Set((Smp::UInt8)   0);       assert(mdkAny.type == Smp::ST_UInt8);
mdkAny.Set((Smp::UInt16)  0);       assert(mdkAny.type == Smp::ST_UInt16);
mdkAny.Set((Smp::UInt32)  0);       assert(mdkAny.type == Smp::ST_UInt32);
mdkAny.Set((Smp::UInt64)  0);       assert(mdkAny.type == Smp::ST_UInt64);
mdkAny.Set((Smp::Float32) 0.0);     assert(mdkAny.type == Smp::ST_Float32);
mdkAny.Set((Smp::Float64) 0.0);     assert(mdkAny.type == Smp::ST_Float64);

mdkAny.Set(dt, Smp::ST_DateTime);   assert(mdkAny.type == Smp::ST_DateTime);
mdkAny.Set(du, Smp::ST_Duration);   assert(mdkAny.type == Smp::ST_Duration);
```

2.4.3 Getter methods

To get the value of an `AnySimple`, the overloaded `Get()` methods can be used. These methods return the value if the `AnySimple` type coincides with the type of the value argument, or throw an exception otherwise.

```
// Get AnySimple
Smp::Char8   char8;   mdkAny.type = Smp::ST_Char8;   mdkAny.Get(char8);
Smp::Bool    boolean; mdkAny.type = Smp::ST_Bool;    mdkAny.Get(boolean);
Smp::Int8    int8;    mdkAny.type = Smp::ST_Int8;    mdkAny.Get(int8);
Smp::Int16   int16;   mdkAny.type = Smp::ST_Int16;   mdkAny.Get(int16);
Smp::Int32   int32;   mdkAny.type = Smp::ST_Int32;   mdkAny.Get(int32);
Smp::Int64   int64;   mdkAny.type = Smp::ST_Int64;   mdkAny.Get(int64);
Smp::UInt8   uint8;   mdkAny.type = Smp::ST_UInt8;   mdkAny.Get(uint8);
Smp::UInt16  uint16;  mdkAny.type = Smp::ST_UInt16;  mdkAny.Get(uint16);
Smp::UInt32  uint32;  mdkAny.type = Smp::ST_UInt32;  mdkAny.Get(uint32);
Smp::UInt64  uint64;  mdkAny.type = Smp::ST_UInt64;  mdkAny.Get(uint64);
Smp::Float32 float32; mdkAny.type = Smp::ST_Float32; mdkAny.Get(float32);
Smp::Float64 float64; mdkAny.type = Smp::ST_Float64; mdkAny.Get(float64);
```

2.4.4 Convert to Smp AnySimple

As `Smp::Mdk::AnySimple` inherits from `Smp::AnySimple`, an Mdk AnySimple can be used whenever an Smp AnySimple is expected.

2.5 String template

```
namespace Smp
{
    namespace Mdk
    {
        /// Template that encapsulates a string.
        template <int size>
        struct String
        {
            Smp::Char8 internalArray[size+1];

            // Constructor
            String()
            {
                memset(internalArray, 0, sizeof(internalArray));
            }
            // Constructor taking a string
            String(Smp::String8 string)
            {
                strncpy(internalArray, string, size);
            }

            // index-get operator
            inline const Smp::Char8& operator [] (long index) const
            {
                assert(index < size);
                return internalArray[index];
            }
            // index-set operator
            inline Smp::Char8& operator [] (long index)
            {
                assert(index < size);
                return internalArray[index];
            }
            // address operator, especially for strings to get a char pointer.
            inline Smp::Char8* operator & (void)
            {
                return internalArray;
            }
            // assignment operator for strings
            inline Smp::Char8* operator = (Smp::String8 string)
            {
                return strncpy(internalArray, string, size);
            }
        };
    }
}
```

The C++ mapping of strings (which are always of a fixed length) is done via a structure with an internal string field, rather than using just an array. This has been done to allow returning strings by value, which avoids questions on where and how to allocate and release memory for strings. On the other hand, it makes working with strings less intuitive. Therefore, the Mdk provides a String template that can be used to create a String type that is compliant with the C++ mapping, and still provides the standard index (`[]`), address (`&`), and assignment (=) operators.

```
    // Define string type using template
    typedef Smp::Mdk::String<32> MyString;

    // Create new string using constructor with string
    MyString myString("Mello World");

    // Fix typo in "Hello" using index operator
    myString[0] = 'H';

    // Print text using address operator
    printf("I say: %s\n", &myString);
```

2.6 Array template

```
namespace Smp
{
    namespace Mdk
    {
        /// Template that encapsulates an array.
        template <class ArrayItemType, int size>
        struct Array
        {
            ArrayItemType internalArray[size];

            // Constructor
            Array()
            {
                memset(internalArray, 0, sizeof(internalArray));
            }

            // index-get operator
            inline const ArrayItemType& operator [] (long index) const
            {
                return internalArray[index];
            }
            // index-set operator
            inline ArrayItemType& operator [] (long index)
            {
                return internalArray[index];
            }
            // address operator, especially for strings to get a char pointer.
            inline ArrayItemType* operator & (void)
            {
                return internalArray;
            }
        };
    }
}
```

The C++ mapping of arrays (which are always of a fixed size) is done via a structure with an internal field, rather than using just an array. This has been done to allow returning arrays by value, which avoids questions on where and how to allocate and release memory for arrays. On the other hand, it makes working with arrays less intuitive.

Therefore, the Mdk provides an Array template that can be used to create an Array type that is compliant with the C++ mapping, and still provides the standard index (`[]`) and address (`&`) operators.

Note that this template can be used to create multi-dimensional arrays as well, as demonstrated in the example below.

```
    // Define one-dimensional array type using template
    typedef Smp::Mdk::Array<Smp::Float64, 3> VectorThreeD;

    // Define two-dimensional array type using template
    typedef Smp::Mdk::Array<VectorThreeD, 3> MatrixThreeD;

    // Create array and two vectors
    VectorThreeD vector, result;
    MatrixThreeD matrix;

    // Calculate result = matrix * vector
    for (int row=0; row<3; row++)
    {
        result[row] = matrix[row][0] * vector[0]
                    + matrix[row][1] * vector[1]
                    + matrix[row][2] * vector[2];
    }
```

3. COMPONENT MODEL

This section details the platform independent component model of SMP2. It contains all types used within the SMP2 Component Model. This includes simple types, derived types, and interfaces with exceptions.

3.1 Objects and Components

In SMP2, a simulation is composed out of components, where models, services, and the simulation environment all implement a common base interface. Other elements in SMP2 are not components, but only objects.

3.1.1 Objects

Objects are the baseline for components. They provide name and description.

3.1.1.1 Implementing IObject

The `Smp::Mdk::Object` class provides an implementation of the `IObject` interface.

To implement `IObject`, derive your class from `Smp::Mdk::Object`, and provide a constructor with name and description that calls the corresponding base constructor of `Smp::Mdk::Object`. Note that the name must be a valid name.

As the SMP interfaces use virtual methods, you always have to provide a virtual destructor as well.

```
class Example : public Smp::Mdk::Object
{
public:
    /// Constructor with name and description.
    /// @param name Name of object.
    /// @param description Description of object.
    Example(Smp::String8 name, Smp::String8 description)
    : Smp::Mdk::Object(name, description)
    {
        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // add your code here
    }
};
```

In addition to implementing the `IObject` interface, the `Smp::Mdk::Object` class provides a static method that evaluates whether a given name is valid.

```
/// Validate the name for rules mentioned at GetName().
/// @param name Name to validate.
/// @returns True if name is valid, false otherwise.
static Smp::Bool ValidateName(Smp::String8 name);
```

This method validates all rules except for the one that names must be unique within their context. As it is a static method, it can be used without having to inherit from the `Smp::Mdk::Object` class.

Several other classes of the MDK are derived from the `Smp::Mdk::Object` class, including `EntryPoint`, `EventSink`, `EventSource`, `Container` and `Reference`.

Inheritance Diagram:

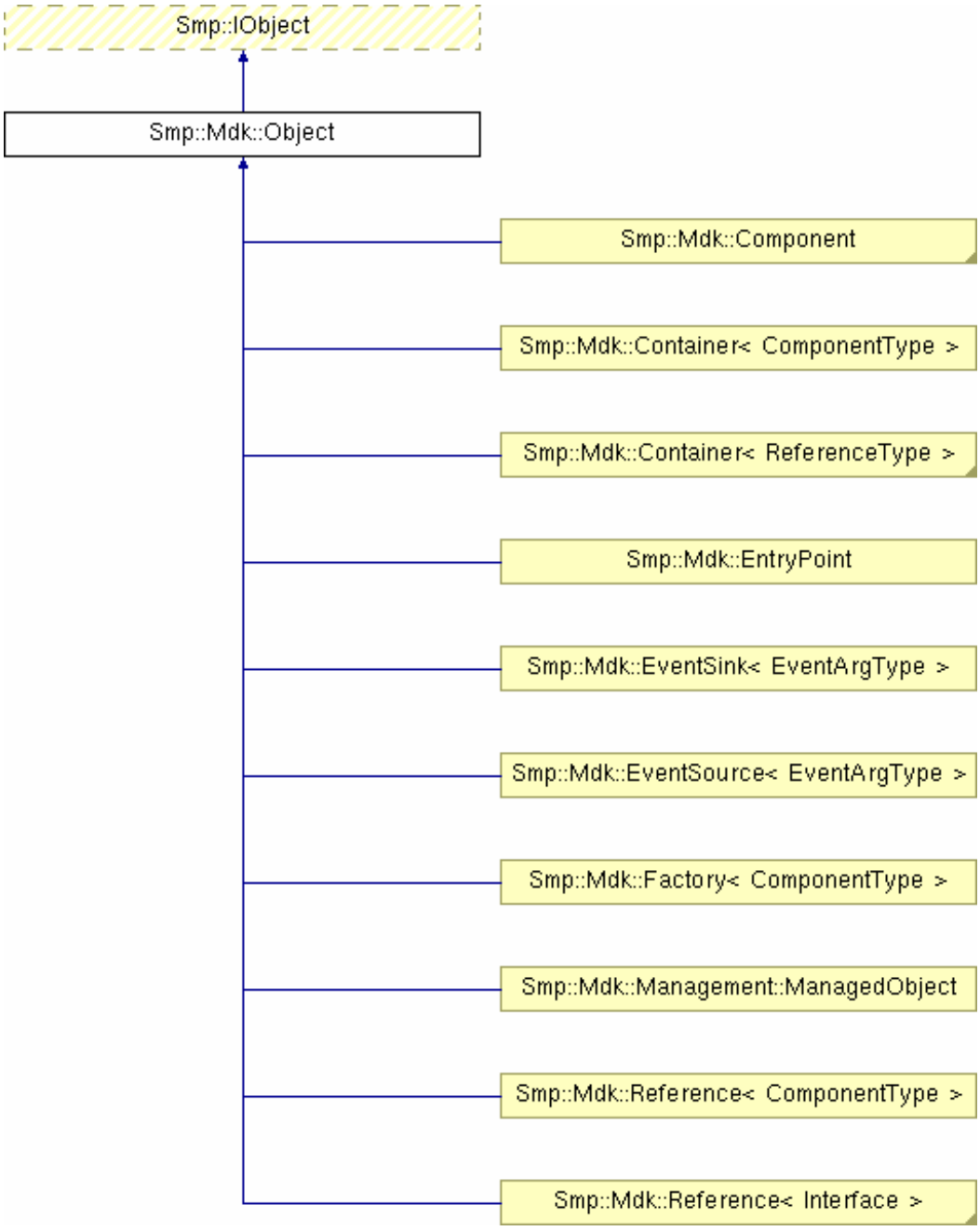

Figure 3-1: Object Inheritance

3.1.2 Components

Most all elements in SMP2 are components, which implement the **`IComponent`** interface.

The three most important component types are models, services, and the simulator, but the MDK only cares about models.

3.1.2.1 Implementing IComponent

The `Smp::Mdk::Component` class provides an implementation of the `IComponent` interface.

To implement `IComponent`, derive your class from `Smp::Mdk::Component`, and provide a constructor with name, description and parent that calls the corresponding base constructor of `Smp::Mdk::Component`. Note that the name must be a valid name.

As the SMP interfaces use virtual methods, you always have to provide a virtual destructor as well.

```
class Example : public Smp::Mdk::Component
{
public:
    /// Constructor with name,description and parent.
    /// @param name Name of component.
    /// @param description Description of component.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Component(name, description, parent)
    {
        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // add your code here
    }
};
```

Inheritance Diagram:

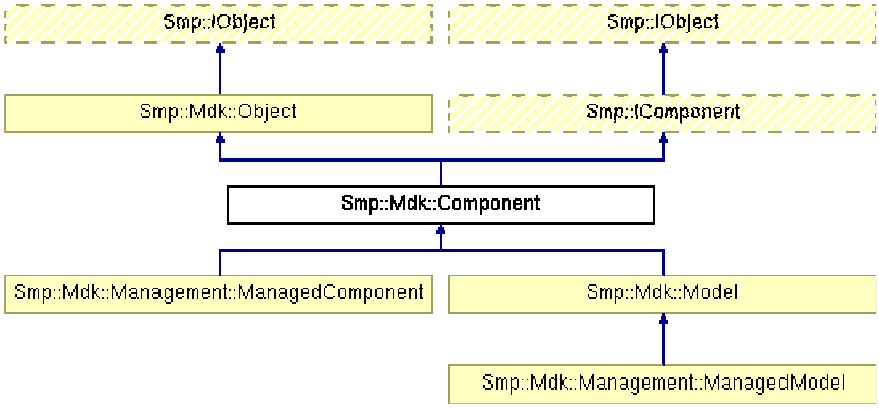

Figure 3-2: Component Inheritance

3.1.2.2 Implementing IModel

The `Smp::Mdk::Model` class provides an implementation of the `IModel` interface.

To implement `IModel`, derive your class from `Smp::Mdk::Model`, and provide a constructor with name, description and parent that calls the corresponding base constructor of `Smp::Mdk::Model`. Note that the name must be a valid name.

As the SMP interfaces use virtual methods, you always have to provide a virtual destructor as well.

The base implementation of `Publish()` and `Connect()` store the given reference into a protected field and change the model state, while the `Configure()` implementation only performs the state transition. Typically, you need to override these methods, but they should call the base implementation to initialise the protected fields, and to perform the state transition.

```
class Example : public Smp::Mdk::Model
{
public:
    /// Constructor with name,description and parent.
    /// @param name Name of component.
    /// @param description Description of component.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Model(name, description, parent)
    {
        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // add your code here
    }

    /// Connect model to the simulator.
    void Connect(Smp::ISimulator* simulator)
    {
        // Call base class implementation first
        Smp::Mdk::Model::Connect(simulator);
        // add your code here
    }

    /// Perform any custom configuration.
    void Configure()
    {
        // add your code here
        // Call base class implementation last
        Smp::Mdk::Model::Configure();
    }

    /// Publish fields to the publication receiver.
    void Publish(Smp::IPublication* receiver)
    {
        // Call base class implementation first
        Smp::Mdk::Model::Publish(receiver);
        // add your code here
    }
};
```

In addition to implementing the `IModel` interface, the `Smp::Mdk::Model` class provides a convenience method that queries for a service by name.

```
/// Get a service by name.
/// Convenience method that delegates the service query to the
/// service provider, which is the simulator passed to Connect().
/// @param   serviceName Service name.
/// @return  Service component.
/// @remarks The returned component may be NULL if no
///          service with the given name could be found.
Smp::IService* GetService(const Smp::String8 serviceName) const;
```

This method can only be called when the `Connect()` method has been called before.

The `Smp::Mdk::Management::ManagedModel` class of the MDK is derived from the `Smp::Mdk::Model` class and adds an implementation for the `IManagedModel` interface.

Inheritance Diagram:

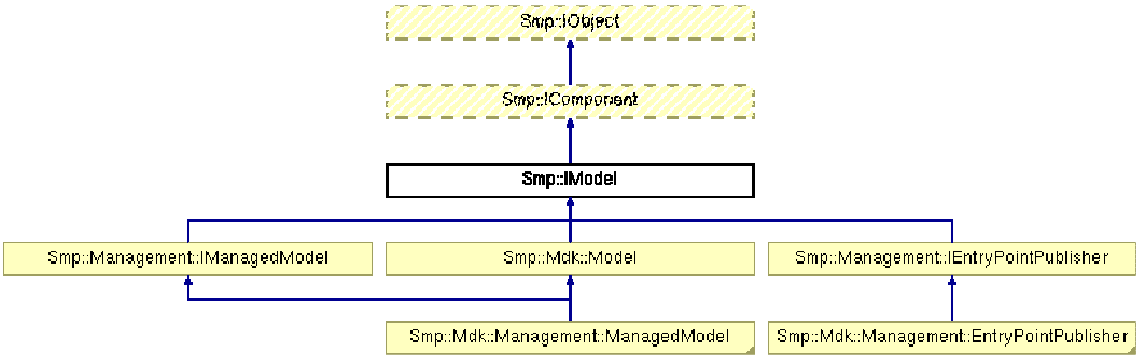

Figure 3-3: Model Inheritance

3.2 Component Mechanisms

While the `IComponent` base interface provides mechanisms to get name, description, and parent, it does not allow specifying further relations between components. The mechanisms supported by SMP2 are aggregation, composition, inter-component events via event sources and event sinks, dynamic invocation and persistence.

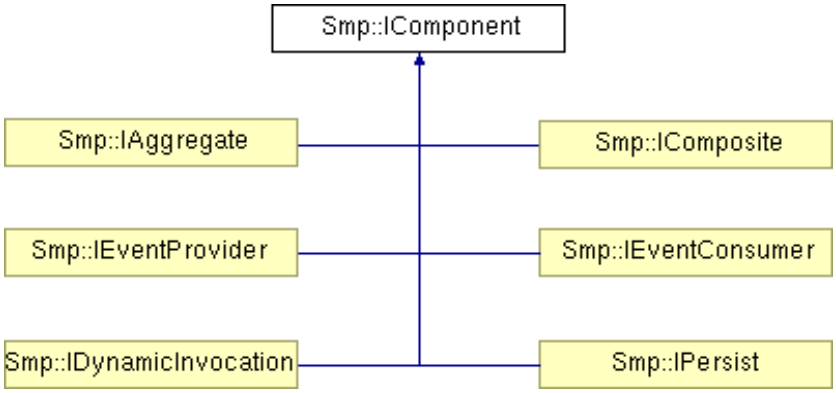

Figure 3-4: Component Mechanisms

3.2.1 Aggregation

Via aggregation, a component can reference other components in the component hierarchy to use their methods. As opposed to composition, an aggregated component is not owned, but only referenced.

3.2.1.1 Implementing IReference

The `Smp::Mdk::Reference` template class provides an implementation of the `IReference` interface. Every reference is typed by an interface, which needs to be passed to the template class constructor to allow for type checking.

To create an object implementing `IReference`, use the constructor of `Smp::Mdk::Reference` with the interface type, and provide name, description and reference provider. Note that the name must be a valid name.

```
public:
    Smp::IReference* MyReference;
...
    // Create a reference
    MyReference = new Smp::Mdk::Reference<Smp::IModel>(
        "MyReference", "This is a reference", this);
```

Inheritance Diagram:

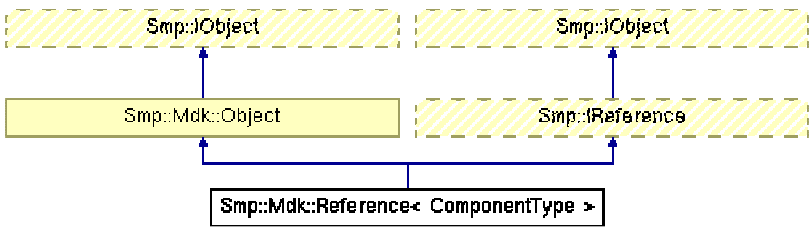

Figure 3-5: Reference Inheritance

References are used together with the `IAggregate` interface for aggregation. It is recommended to always use `Smp::Mdk::Management::ManagedReference`, which is the managed version of this template class and derived from it.

3.2.1.2 Implementing IAggregate

As aggregation is a component mechanism, it can only be implemented by components. Typically, these components are models, but the example shown here is a component only, to keep the source code small.

The `Smp::Mdk::Aggregate` class provides an implementation of the `IAggregate` interface.

To implement `IAggregate`, derive your component from `Smp::Mdk::Aggregate` as well, and add all local references via the convenience method `AddReference()`.

```
class Example :
    public Smp::Mdk::Component,
    public virtual Smp::Mdk::Aggregate
{
public:
    Smp::IReference* MyReference;

    /// Constructor with name,description and parent.
    /// @param name Name of component.
    /// @param description Description of component.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Component(name, description, parent)
    {
        // Create a reference
        MyReference = new Smp::Mdk::Reference<Smp::IModel>(
            "MyReference", "This is a reference", this);

        // Expose the reference for aggregation
        this->AddReference(MyReference);

        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
         // Delete reference
         delete MyReference;

        // add your code here
    }
};
```

Inheritance Diagram:

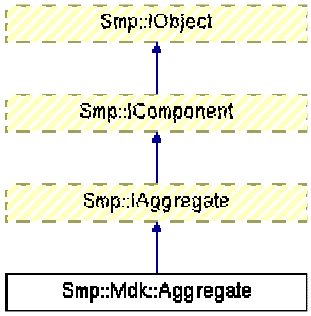

Figure 3-6: Aggregate Inheritance

3.2.2 Composition

Via composition, a component can contain other components in the component hierarchy. As opposed to aggregation, a component is owned, and its life-time coincides with its parent component. Composition is the counter-part to the `GetParent()` method of the `IComponent` interface and allows traversing the tree of components in any direction.

3.2.2.1 Implementing IContainer

The `Smp::Mdk::Container` template class provides an implementation of the `IContainer` interface. Every container is typed by a reference type (an interface or model), which needs to be passed to the template class constructor to allow for type checking.

To create an object implementing `IContainer`, use the constructor of `Smp::Mdk::Container` with the reference type, and provide name, description and parent composite. Note that the name must be a valid name.

```
public:
    Smp::IContainer* MyContainer;
...
    // Create a container
    MyContainer = new Smp::Mdk::Container<Smp::IModel>(
        "MyContainer", "This is a container", this);
```

Inheritance Diagram:

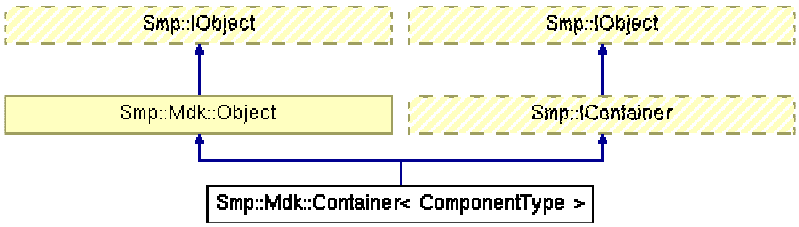

Figure 3-7: Container Inheritance

Containers are used together with the `IComposite` interface for composition. It is recommended to always use `Smp::Mdk::Management::ManagedContainer`, which is the managed version of this template class and derived from it.

3.2.2.2 Implementing IComposite

As composition is a component mechanism, it can only be implemented by components. Typically, these components are models, but the example shown here is a component only, to keep the source code small.

The `Smp::Mdk::Composite` class provides an implementation of the `IComposite` interface.

To implement `IComposite`, derive your component from `Smp::Mdk::Composite` as well, and add all local references via the convenience method `AddContainer()`.

```
class Example :
    public Smp::Mdk::Component,
    public virtual Smp::Mdk::Composite
{
public:
    Smp::IContainer* MyContainer;

    /// Constructor with name,description and parent.
    /// @param name Name of component.
    /// @param description Description of component.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Component(name, description, parent)
    {
        // Create a container
        MyContainer = new Smp::Mdk::Container<Smp::IModel>(
            "MyContainer", "This is a container", this);

        // Expose the container for composition
        this->AddContainer(MyContainer);

        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // Delete container
        delete MyContainer;

        // add your code here
    }
};
```

Inheritance Diagram:

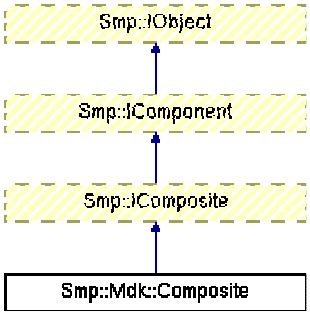

Figure 3-8: Composite Inheritance

3.2.3 Events

Events are used in event-based programming. Event-based programming works via event sources and event sinks that can be registered to and unregistered from event sources. When an event source emits an event, it notifies all subscribed event sinks.

In SMP2, an event sink is an interface with a `Notify()` method, which takes an event argument of type `AnySimple`. The MDK supports an event handler with a typed event argument, which can be any of the pre-defined simple types. Correspondingly, the `Subscribe()` method of an event source validates that the subscribed event sink is of the same type. As this is not a required feature, it only works when both the event source and the event sink use the MDK templates for their implementation. Otherwise, no type checking of the event argument can be performed.

To be able to find out whether a given event sink is using the MDK implementation, the additional class `Smp::Mdk::MdkEventSink` has been introduced. It does not add any functionality, but allows to find out (via `dynamic_case<Smp::Mdk::MdkEventSink >`) whether an event sink uses the MDK.

Inheritance Diagram:

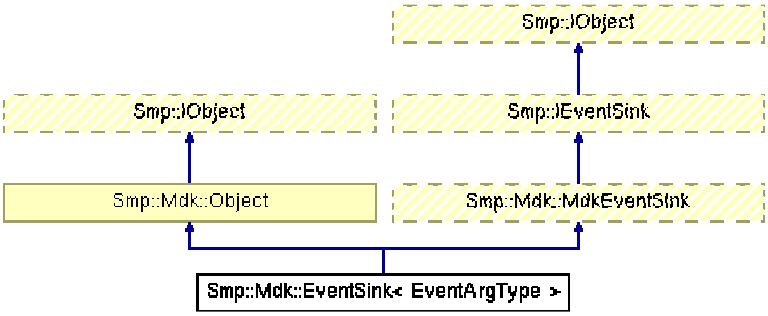

Figure 3-9: EventSink Inheritance

3.2.3.1 Implementing IEventSink

The `Smp::Mdk::EventSink` template class provides an implementation of the `IEventSink` interface for an event handler with a typed event argument.

To create an object implementing `IEventSink`, use the constructor of `Smp::Mdk::EventSink` with the event argument type, and provide name, description, event consumer and event handler. Note that the name must be a valid name. The event handler must have an event argument of the specified type. Otherwise, an exception is thrown.

```
class Example : public Smp::Mdk::Component
{
private:
    /// Event sink handler with typed event argument.
    void MyEventSinkHandler(Smp::IObject* sender, Smp::Bool arg)
    {
        // add your code here
    }

public:
    /// Event sink interface.
    Smp::IEventSink* MyEventSink;

    /// Constructor with name,description and parent.
    /// @param name Name of component.
    /// @param description Description of component.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Component(name, description, parent)
    {
        // Create an event sink
        MyEventSink = new Smp::Mdk::EventSink<Smp::Bool>(
            "MyEventSink",                  // Name
            "This is an event sink",        // Description
            this,                           // Event consumer
            &Example::MyEventSinkHandler);  // Event handler

        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // Delete event sink
        delete MyEventSink;

        // add your code here
    }
};
```

The `Smp::Mdk::VoidEventSink` template class provides an implementation of the `IEventSink` interface for an event handler without a typed event argument.

To create an object implementing `IEventSink`, use the constructor of `Smp::Mdk::VoidEventSink` and provide name, description, event consumer and event handler. Note that the name must be a valid name. The event handler must not have an event argument. Otherwise, an exception is thrown.

```
class Example : public Smp::Mdk::Component
{
private:
    /// Event sink handler without event argument.
    void MyVoidEventSinkHandler(Smp::IObject* sender)
    {
        // add your code here
    }

public:
    /// Event sink interface.
    Smp::IEventSink* MyVoidEventSink;

    /// Constructor with name,description and parent.
    /// @param name Name of component.
    /// @param description Description of component.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Component(name, description, parent)
    {
        // Create an event sink
        MyVoidEventSink = new Smp::Mdk::VoidEventSink (
            "MyVoidEventSink",                  // Name
            "This is a void event sink",        // Description
            this,                               // Event consumer
            &Example::MyVoidEventSinkHandler);  // Event handler

        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // Delete event sink
        delete MyVoidEventSink;

        // add your code here
    }
};
```

3.2.3.2 Implementing IEventSource

The `Smp::Mdk::EventSource` template class provides an implementation of the `IEventSource` interface for an event source with a typed event argument.

To create an object implementing `IEventSource`, use the constructor of `Smp::Mdk::EventSource` with the event argument type, and provide name, description, event consumer and event handler. Note that the name must be a valid name.

```
class Example : public Smp::Mdk::Component
{
public:
    /// Event source interface.
    Smp::IEventSource* MyEventSource;

    /// Constructor with name,description and parent.
    /// @param name Name of component.
    /// @param description Description of component.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Component(name, description, parent)
    {
        // Create an event source
        MyEventSource = new Smp::Mdk::EventSource<Smp::Bool>(
            "MyEventSource",                  // Name
            "This is an event source",        // Description
            this);                            // Event provider

        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // Delete event source
        delete MyEventSource;

        // add your code here
    }
};
```

Inheritance Diagram:

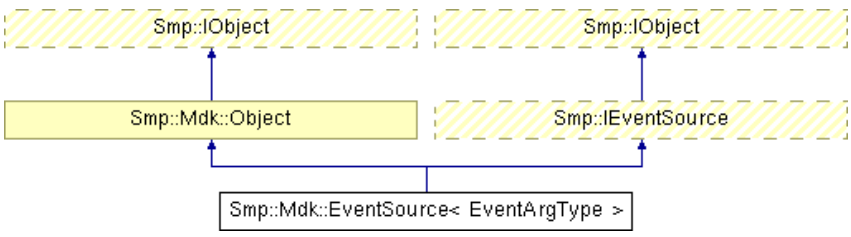

Figure 3-10: EventSource Inheritance

The `Smp::Mdk::VoidEventSource` template class provides an implementation of the `IEventSource` interface for an event source without a typed event argument.

To create an object implementing `IEventSource` for an event source without event argument, use the constructor of `Smp::Mdk::VoidEventSource`, and provide name, description, event consumer and event handler. Note that the name must be a valid name.

```
class Example : public Smp::Mdk::Component
{
public:
    /// Event source interface.
    Smp::IEventSource* MyVoidEventSource;

    /// Constructor with name,description and parent.
    /// @param name Name of component.
    /// @param description Description of component.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Component(name, description, parent)
    {
        // Create an event source
        MyVoidEventSource = new Smp::Mdk::VoidEventSource(
            "MyVoidEventSource",              // Name
            "This is a void event source",    // Description
            this);                            // Event provider

        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // Delete event source
        delete MyVoidEventSource;

        // add your code here
    }
};
```

To emit an event (i.e. call all connected event sinks) of an event source with an event argument, the `EventSource` MDK template class provides a typed `Emit()` method which can be called with a sender (an `IObject`) and an event argument.

```
(dynamic_cast<Smp::Mdk::EventSource<Smp::Bool>*>(MyEventSource))->Emit(this, false);
```

To emit an event (i.e. call all connected event sinks) of an event source without an event argument, the `VoidEventSource` MDK template class provides an `Emit()` method which can be called with a sender (an `IObject`).

```
(dynamic_cast<Smp::Mdk::VoidEventSource*>(MyVoidEventSource))->Emit(this);
```

3.2.4 Dynamic Invocation

Dynamic invocation is a mechanism that makes the operations of a component available via a standardised interface (as opposed to a custom interface of the component which is not known at compile time of the simulation environment). In order to allow calling a named method with any number of parameters, a request object has to be created which contains all information needed for the method invocation. This request object is as well used to transfer back a return value of the operation.

The dynamic invocation concept of SMP2 standardises the request objects (`IRequest` interface). In addition, two methods are provided as part of `IDynamicInvocation` to create and delete request objects. However, it is not mandatory to use these methods, as request objects can well be created and deleted using another implementation.

3.2.4.1 Implementing IDynamicInvocation

No reference implementation of `IDynamicInvocation` is available, as this is model specific. However, the MDK does recommend how to implement two of the three methods of `IDynamicInvocation`.

Every model is asked to publish its fields and operations to an instance implementing `IPublication` in its `Publish()` method. If all operations that shall be available for dynamic invocation are published that way, the two methods `CreateRequest()` and `DeleteRequest()` can be delegated to the `IPublication` interface. This works for models derived from `Smp::Mdk::Model` that call the base implementation of `Publish()`.

```
class Example :
    public Smp::Mdk::Model,
    public virtual Smp::IDynamicInvocation
{
public:
    /// Publish fields to the publication receiver.
    void Publish(Smp::IPublication* receiver)
    {
        // Call base class implementation first
        Smp::Mdk::Model::Publish(receiver);

        // add your code here
    }

    /// Returns a request for the given operation
    Smp::IRequest* CreateRequest(Smp::String8 operationName)
    {
        // Delegate implementation to IPublication
        return this->m_publication->CreateRequest(operationName);
    }

    /// Delete a request
    void DeleteRequest(Smp::IRequest *request)
    {
        // Delegate implementation to IPublication
        this->m_publication->DeleteRequest(request);
    }
};
```

The implementation of `Invoke()` has to be done manually.

3.2.4.2 Implementing IRequest

Typically, there is no need for a model to implement this interface, as creating request objects of methods that have been published can be delegated to the `IPublication` interface. However, for methods not published, the MDK helper class can be useful.

The `Smp::Mdk::Request` class provides an implementation of the `IRequest` interface. It provides not only the methods of this interface, but an additional method to add a parameter.

To create an object implementing `IRequest`, use the constructor of `Smp::Mdk::Request` and provide operation name and return type. Note that the name must be a valid name.

```
Smp::IRequest* CreateRequest(Smp::String8 operationName)
{
    if (strcmp(operationName, "MyEntryPoint") == 0)
    {
        Smp::Mdk::Request* request = new Smp::Mdk::Request(
            operationName, Smp::ST_None);

        return request;
    }
    else
    {
        return NULL;
    }
}
```

As the next step, you need to add the parameters in their order, using the method `AddParameter()`.

```
request->AddParameter("BoolParam", Smp::ST_Bool);
request->AddParameter("Float64Param", Smp::ST_Float64);
```

3.2.5 Persistence

Persistence of SMP2 components can be handled in one of two ways:

1. **External Persistence**: The simulation environment stores and restores the model's state by directly accessing the fields that are published to the simulation environment, i.e. via the `IPublication` interface.

   Note: This should be the preferred mechanism for the majority of models.

2. **Self-Persistence**: The component *may* implement the `IPersist` interface, which allows it to store and restore (part of) its state into or from storage that is provided by the simulation environment.

   Note: This mechanism is usually only needed by specialised models, for example embedded models that need to load on-board software from a specific file. Further, this mechanism can be used by simulation services if desired. For example, the Scheduler service may use it to store and restore its current state.

Like all features in this section, self-persistence of models and components is an optional feature, while external persistence (via the Store and Restore methods of the **`ISimulator`** interface) is a mandatory feature of every SMP2 simulation environment.

3.2.5.1 Implementing IPersist

As the purpose of `IPersist` is to allow custom serialisation of specialised models, the MDK cannot provide a base implementation of this interface.

3.3 Model Mechanisms

While the `IModel` interface defines the mandatory functionality every SMP2 model has to provide, this section introduces additional mechanisms available for more advanced use.

3.3.1 Entry Points

An entry point is an interface that exposes a void function with no return value that can be called by the scheduler or event manager service.

3.3.1.1 Implementing IEntryPoint

The `Smp::Mdk::EntryPoint` template class provides an implementation of the `IEntryPoint` interface.

To create an object implementing `IEntryPoint`, use the constructor of `Smp::Mdk::EntryPoint` with name, description, publisher and the internal method implementing the entry point. Note that the name must be a valid name, and the publisher which owns the entry point need to implement `IComponent`.

```
class Example : public Smp::Mdk::Model
{
private:
    // This is the implementation of the entry point.
    void MyEntryPointImplementation(void)
    {
        // add your code here
    }

public:
    /// Entry point interface.
    Smp::IEntryPoint* MyEntryPoint;

    /// Constructor with name,description and parent.
    /// @param name Name of component.
    /// @param description Description of component.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Model(name, description, parent)
    {
        // Create an entry point
        MyEntryPoint = new Smp::Mdk::EntryPoint(
            "MyEntryPoint",                     // Name
            "This is an entry point",           // Description
            this,                               // Publisher
            &Example::MyEntryPointImplementation);  // Implementation

        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // Delete entry point
        delete MyEntryPoint;

        // add your code here
    }
};
```

Inheritance Diagram:

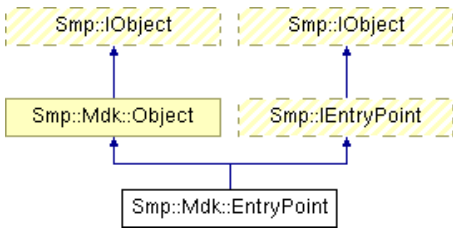

Figure 3-11: EntryPoint Inheritance

3.4 Management Interfaces

Managed interfaces allow external components to access all mechanisms by name. This includes the basic component features, optional component mechanisms and optional model mechanisms.

Managed interfaces allow full access to all functionality of components. For composition and aggregation, they extend the existing interfaces by methods to add new components or references, respectively. For entry points, event sources and event sinks, the managed interfaces provide access to the elements by name. For fields, access by name is provided by an extended interface allowing reading and writing field values.

All management interfaces are optional, and only need to be provided for models used in a managed environment. Typically, in a managed environment a model configuration is build from an XML document (namely an SMDL Assembly) during the `Creating` phase. However, the MDK makes implementing these interfaces very easy, so it is recommended to always implement them.

3.4.1 Managed Components

Managed components provide write access to their properties, i.e. they provide corresponding "setter" methods for the `Name`, `Description`, and `Parent` properties. This allows putting them into a hierarchy with a given name and description.

3.4.1.1 Implementing IManagedObject

The `Smp::Mdk::Management::ManagedObject` class provides an implementation of the `IManagedObject` interface.

To implement `IManagedObject`, derive from `Smp::Mdk::Management::ManagedObject`, and provide at least an empty constructor. In addition you may provide a constructor with name and description that calls the corresponding base constructor of `Smp::Mdk::Management::ManagedObject`. Note that the name must be a valid name.

As the SMP interfaces use virtual methods, you always have to provide a virtual destructor as well.

```
class Example : public Smp::Mdk::Management::ManagedObject
{
public:
    /// Default constructor.
    Example()
    {
        // add your code here
    }

    /// Constructor with name and description.
    /// @param name Name of object.
    /// @param description Description of object.
    Example(Smp::String8 name, Smp::String8 description)
    : Smp::Mdk::Management::ManagedObject(name, description)
    {
        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // add your code here
    }
};
```

Within the implementation, name and description can be accessed via the protected fields `m_name` and `m_description`.

Inheritance Diagram:

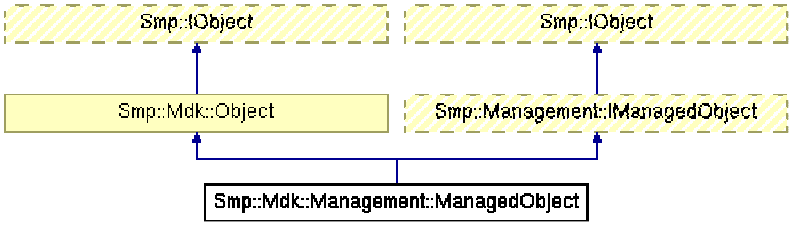

Figure 3-12: ManagedObject Inheritance

3.4.1.2 Implementing IManagedComponent

The `Smp::Mdk::Management::ManagedComponent` class provides an implementation of the `IManagedComponent` interface.

To implement `IManagedComponent`, derive your class from the MDK base class `Smp::Mdk::Management::ManagedComponent`, and provide at least an empty constructor. In addition you may provide a constructor with name, description and parent that calls the corresponding base constructor of `Smp::Mdk::Management::ManagedComponent`. Note that the name must be a valid name.

As the SMP interfaces use virtual methods, you always have to provide a virtual destructor as well.

```
class Example : public Smp::Mdk::Management::ManagedComponent
{
public:
    /// Default constructor.
    Example()
    {
        // add your code here
    }

    /// Constructor with name and description.
    /// @param name Name of object.
    /// @param description Description of object.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Management::ManagedComponent(name, description, parent)
    {
        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // add your code here
    }
};
```

Within the implementation, name, description and parent can be accessed via the protected fields `m_name`, `m_description` and `m_parent`.

Inheritance Diagram:

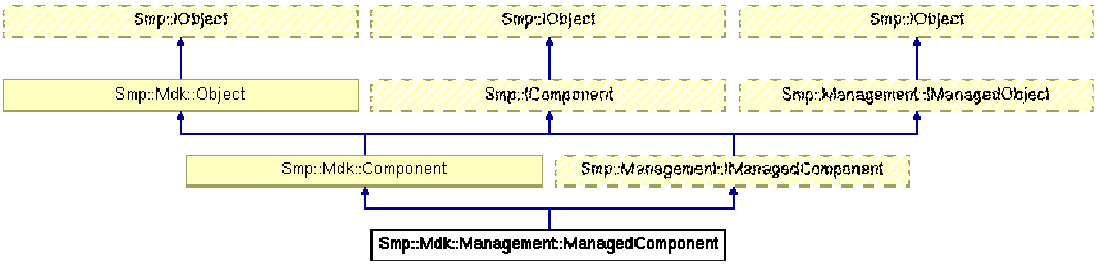

Figure 3-13: ManagedComponent Inheritance

3.4.2 Managed Component Mechanisms

The component mechanisms introduced in 3.2 (Component Mechanisms) do not provide full access for external components, but only limited access:

- Aggregation provides collections of references, but does not allow adding new references to a Reference.
- Composition provides a tree of components, but does not allow adding new components to a Container.
- Event sinks can be connected by models if they have access to event sources, but an external component can not query for event sinks and event sources by name.

To overcome these limitations, managed interfaces are provided with full access to all functionality. For composition and aggregation, these extend the existing interfaces by methods to add new components or references, respectively. For event sources and event sinks, the managed interfaces provide access to the elements by name.

3.4.2.1 Implementing IManagedReference

The `Smp::Mdk::Management::ManagedReference` template class provides an implementation of the `IManagedReference` interface. Every reference is typed by an interface, which needs to be passed to the template class constructor to allow for type checking.

To create an object implementing the `IManagedReference` interface, use the constructor of `Smp::Mdk::Management::ManagedReference` with the interface type, and provide name, description, reference provider and multiplicity. Note that the name must be a valid name.

```
public:
    Smp::IManagedReference* MyReference;
...
    // Create a managed reference
    MyReference = new Smp::Mdk::Management::ManagedReference<Smp::IModel>(
        "MyReference", "This is a reference", this, 0, -1);
```

Inheritance Diagram:

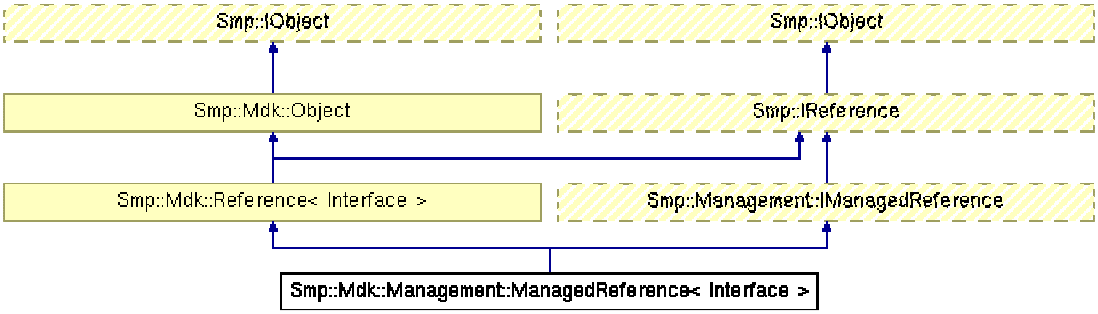

Figure 3-14: ManagedReference Inheritance

3.4.2.2 Implementing IManagedContainer

The `Smp::Mdk::Management::ManagedContainer` template class provides an implementation of the `IManagedContainer` interface. Every container is typed by a reference type (an interface or model), which needs to be passed to the template class constructor to allow for type checking.

To create an object implementing the `IManagedContainer` interface, use the constructor of `Smp::Mdk::Management::ManagedContainer` with the reference type, and provide name, description, parent composite and multiplicity. Note that the name must be a valid name.

```
public:
    Smp::IManagedContainer* MyContainer;
...
    // Create a managed container
    MyContainer = new Smp::Mdk::Management::ManagedContainer<Smp::IModel>(
        "MyContainer", "This is a container", this, 0, -1);
```

Inheritance Diagram:

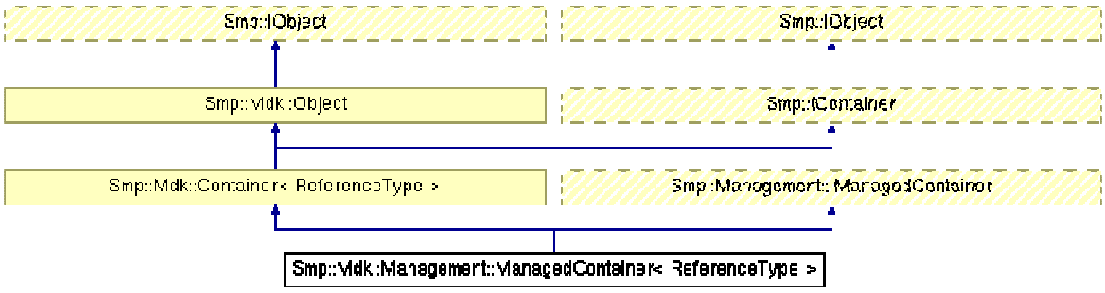

Figure 3-15: ManagedContainer Inheritance

3.4.2.3 Implementing IEventConsumer

As consuming events is a component mechanism, it can only be implemented by components. Typically, these components are models, but the example shown here is a component only, to keep the source code small.

The `Smp::Mdk::Management::EventConsumer` class provides an implementation of the `IEventConsumer` interface.

To implement `IEventConsumer`, derive from `Smp::Mdk::Management::EventConsumer` as well, and add all event sinks via the convenience method `AddEventSink()`.

```
class Example :
    public Smp::Mdk::Component,
    public virtual Smp::Mdk::Management::EventConsumer
{
private:
    /// Event sink handler with typed event argument.
    void MyEventSinkHandler(Smp::IObject* sender, Smp::Bool arg)
    {
        // add your code here
    }

public:
    /// Event sink interface.
    Smp::IEventSink* MyEventSink;

    /// Constructor with name,description and parent.
    /// @param name Name of component.
    /// @param description Description of component.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Component(name, description, parent)
    {
        // Create an event sink
        MyEventSink = new Smp::Mdk::EventSink<Smp::Bool>(
            "MyEventSink",                  // Name
            "This is an event sink",        // Description
            this,                           // Event consumer
            &Example::MyEventSinkHandler);  // Event handler

        // Expose the event sink for event consumer
        this->AddEventSink(MyEventSink);

        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // Delete event sink
        delete MyEventSink;

        // add your code here
    }
};
```

Inheritance Diagram:

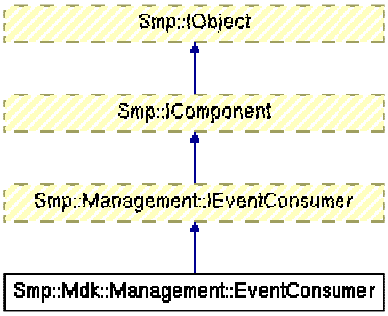

Figure 3-16: EventConsumer Inheritance

Remarks:

This is an optional interface. It needs to be implemented for managed components only, which want to allow access to event sinks by name. However, the overhead of implementing this interface when using the MDK is minimal, so it is strongly recommended to always implement it.

3.4.2.4 Implementing IEventProvider

As providing events is a component mechanism, it can only be implemented by components. Typically, these components are models, but the example shown here is a component only, to keep the source code small.

The `Smp::Mdk::Management::EventProvider` class provides an implementation of the `IEventProvider` interface.

To implement `IEventProvider`, derive from `Smp::Mdk::Management::EvenProvider` as well, and add all event sources via the convenience method `AddEventSource()`.

```
class Example :
    public Smp::Mdk::Component,
    public virtual Smp::Mdk::Management::EventProvider
{
public:
    /// Event source interface.
    Smp::IEventSource* MyEventSource;

    /// Constructor with name,description and parent.
    /// @param name Name of component.
    /// @param description Description of component.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Component(name, description, parent)
    {
        // Create an event source
        MyEventSource = new Smp::Mdk::EventSource<Smp::Bool>(
            "MyEventSource",                // Name
            "This is an event source",      // Description
            This);                          // Event consumer

        // Expose the event source for event provider
        this->AddEventSource(MyEventSource);

        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // Delete event source
        delete MyEventSource;

        // add your code here
    }

    /// Sample function that shows how to emit the event.
    void EmitEvent(Smp::Bool arg)
    {
        // As an example, emit an event to my event source
        (dynamic_cast<Smp::Mdk::EventSource<Smp::Bool>*>(MyEventSource))
         ->Emit(this, arg);
    }
};
```

Inheritance Diagram:

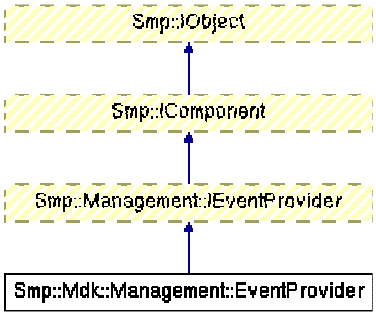

Figure 3-17: EventProvider Inheritance

Remarks:

This is an optional interface. It needs to be implemented for managed components only, which want to allow access to event sources by name. However, the overhead of implementing this interface when using the MDK is minimal, so it is strongly recommended to always implement it.

3.4.3 Managed Model Mechanisms

The model mechanisms introduced in 3.3 (Model Mechanisms) do not provide full access for external components, but only limited access:

- Entry points can be registered with services by models, but an external component cannot query for entry points by name.
- Fields are published against the simulation environment, but no read and write access to named fields is provided.

To overcome these limitations, managed interfaces are provided with full access to all functionality. For entry points, the managed interface provides access to the entry points by name. For fields, access by name is provided by an extended interface allowing reading and writing field values.

3.4.3.1 Implementing IManagedModel

The `Smp::Mdk::Management::ManagedModel` class provides an implementation of the `IManagedModel` interface.

To implement `IManagedModel`, derive from `Smp::Mdk::Management::ManagedModel`, and provide at least an empty constructor. In addition you may provide a constructor with name and description that calls the corresponding base constructor of `Smp::Mdk::Management::ManagedModel`. Note that the name must be a valid name.

As for implementing `IModel`, you will typically need to overload the `Publish()`, `Configure()` and `Connect()` methods. When doing so, make sure you call the base implementation. This is essential as several calls of the `IManagedModel` interface are delegated to the `IPublication` interface.

As the SMP interfaces use virtual methods, you always have to provide a virtual destructor as well.

```
class Example : public Smp::Mdk::Management::ManagedModel
{
public:
    /// Default constructor.
    Example()
    {
        // add your code here
    }

    /// Constructor with name and description.
    /// @param name Name of object.
    /// @param description Description of object.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Management::ManagedModel(name, description, parent)
    {
        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // add your code here
    }

    /// Publish fields to the publication receiver.
    void Publish(Smp::IPublication* receiver)
    {
        // Call base class implementation first
        Smp::Mdk::Model::Publish(receiver);

        // add your code here
    }
};
```

Inheritance Diagram:

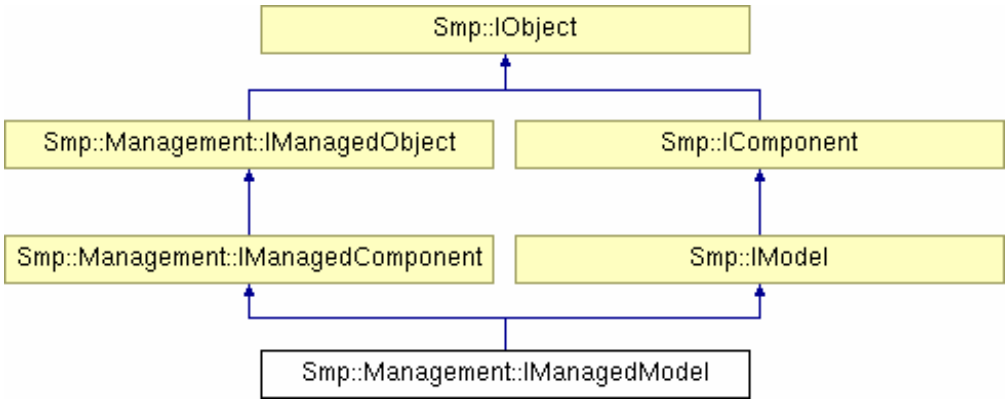

Figure 3-18: IManagedModel Inheritance

3.4.3.2 Implementing IEntryPointPublisher

As publishing entry points is a model mechanism, only models can implement it.

The `Smp::Mdk::Management:EntryPointPublisher` class provides an implementation of the `IEntryPointPublisher` interface.

To implement `IEntryPointPublisher`, derive your model not only from `Smp::Mdk::Model`, but from `Smp::Mdk::Management::EntryPointPublisher` as well, and add all entry points via the convenience method `AddEntryPoint()`.

```
class Example :
    public Smp::Mdk::Model,
    public virtual Smp::Mdk::Management::EntryPointPublisher
{
private:
    // This is the implementation of the entry point.
    void MyEntryPointImplementation(void)
    {
        // add your code here
    }

public:
    /// Entry point interface.
    Smp::IEntryPoint* MyEntryPoint;

    /// Constructor with name,description and parent.
    /// @param name Name of component.
    /// @param description Description of component.
    /// @param parent Parent of component.
    Example(Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent)
    : Smp::Mdk::Model(name, description, parent)
    {
        // Create an entry point
        MyEntryPoint = new Smp::Mdk::EntryPoint(
            "MyEntryPoint",                          // Name
            "This is an entry point",                // Description
            this,                                    // Publisher
            &Example::MyEntryPointImplementation);  // Implementation

        // Add entry point to entry point publisher
        this->AddEntryPoint(MyEntryPoint);

        // add your code here
    }

    /// Virtual destructor.
    virtual ~Example()
    {
        // Delete entry point
        delete MyEntryPoint;

        // add your code here
    }
};
```

Inheritance Diagram:

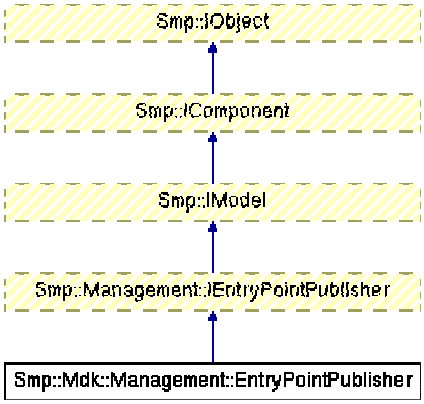

Figure 3-19: EntryPointPublisher Inheritance

Remarks:

This is an optional interface. It needs to be implemented for managed models only. However, the overhead of implementing this interface when using the MDK is minimal, so it is strongly recommended to always implement it.

3.5 Component Factories

Although the development of a simulation environment is not supported by the model development kit, the Mdk provides an implementation for the `IFactory` interface. Every model participating in a dynamically configured simulation has to provide such a factory, which can easily be created with the `Factory` template class.

3.5.1.1 Implementing IFactory

The `Smp::Mdk::Factory` template class provides an implementation of the `IFactory` interface. Every factory is typed by a component type, which needs to be passed to the template class constructor.

To create a factory implementing `IFactory`, use the constructor of `Smp::Mdk::Factory` with the component type, and provide name, description, specification Uuid and implementation Uuid. Note that the name must be a valid name, and that the implementation Uuid needs to be unique.

```
Smp::IFactory* factory;

factory = new Smp::Mdk::Factory<Examples::MdkManagedSample>(
    "MdkManagedSample",
    "Managed Model developed using the MDK",
    specificationUuid,
    implementationUuid);
```

The `IFactory` interface is not implemented by the model or one of its objects, but by the corresponding component factory. This factory is typically needed in the `Initialise()` function in a Dynamic Link Library (**DLL**) or Dynamic Shared Object (**DSO**), where all component factories are registered with the dynamic simulator.

Inheritance Diagram:

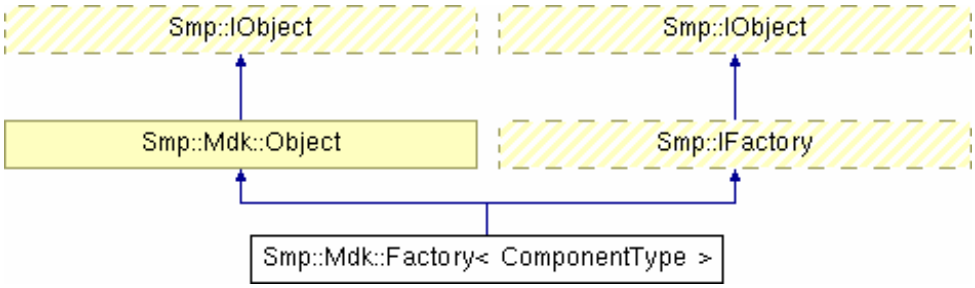

Figure 3-20: Factory Inheritance

4. EXAMPLE MODELS

This section puts the individual concepts into context by showing how to implement a reference model using the Mdk.

4.1 The Example Model

All examples in this section are based on a sample model with the following features:

- It provides self-persistence (`IPersist`)
- It provides one reference named "`MyReference`" (`IReference`), and hence has to implement aggregation (`IAggregate`)
- It provides one container named "`MyContainer`" (`IContainer`), and hence has to implement composition (`IComposite`)
- It provides one entry point named "`MyEntryPoint`" (`IEntryPoint`)
- It provides one event source named "`MyEventSource`" (`IEventSource`)
- It provides one event sink named "`MyEventSink`" (`IEventSink`)
- It implements dynamic invocation (`IDynamicInvocation`)

Further, the model demonstrate how other interfaces and mechanisms can be used:

- It uses the Logger (`ILogger`) to log a message
- It uses the Time Keeper (`ITimeKeeper`) to get epoch time
- It uses the Scheduler (`IScheduler`) to schedule its entry point
- It uses the Publication mechanism (`IPublication`) to publish one field for each of the primitive types
- It shows how to emit an event by calling all event sinks connected to its event source in its entry point implementation
- It shows how to handle an event by writing a message to the console when its event sink is called

This section presents four different models with this functionality: Two models use the Mdk only for the implementation, not for the definition (i.e. no includes of Mdk header files), while the other two models are derived from existing Mdk classes. Further, each of these two cases is implemented both as an unmanaged and as a managed model.

The Mdk does not support certain functionality of the example:

- Self-persistence is a custom feature and not supported
- Dynamic invocation is a custom feature and only supported by the `Request` class
- The implementation of the behaviour of entry points has to be done manually
- The implementation of the behaviour of event sinks has to be done manually

Therefore, these features are implemented in the base class already, while the four example models focus on demonstrating how to make best use of Mdk base classes and template classes in the implementation.

4.1.1 Definition of Sample

```
#ifndef EXAMPLES_H_
#define EXAMPLES_H_

// ----------------------------------------------------------------------------
// ---------------------------- Include files ---------------------------------
// ----------------------------------------------------------------------------

#include "Smp/SimpleTypes.h"
#include "Smp/IModel.h"
#include "Smp/IPersist.h"
#include "Smp/ISimulator.h"
#include "Smp/IComposite.h"
#include "Smp/IContainer.h"
#include "Smp/IAggregate.h"
#include "Smp/IReference.h"
#include "Smp/IEntryPoint.h"
#include "Smp/IEventSink.h"
#include "Smp/IEventSource.h"
#include "Smp/IDynamicInvocation.h"

// ----------------------------------------------------------------------------
// ---------------------------- Class Sample ----------------------------------
// ----------------------------------------------------------------------------

namespace Examples
{
    /// Size of buffer to use for implementation of IPersist.
    const long BUFFER_SIZE = 256;

    /// Name of the Reference that the example models expose
    const Smp::String8 ReferenceName    = "MyReference";

    /// Name of the Container that the example models expose
    const Smp::String8 ContainerName    = "MyContainer";

    /// Name of the Entry Point that the example models expose
    const Smp::String8 EntryPointName   = "MyEntryPoint";

    /// Name of the Event Sink that the example models expose
    const Smp::String8 EventSinkName    = "MyEventSink";

    /// Name of the Event Source that the example models expose
    const Smp::String8 EventSourceName  = "MyEventSource";

    /// This base class defines the functionality to implement by the models.
    class Sample :
        public virtual Smp::IModel,
        public virtual Smp::IPersist,
        public virtual Smp::IAggregate,
        public virtual Smp::IComposite,
        public virtual Smp::IDynamicInvocation
    {
    private:
        int m_buffer[BUFFER_SIZE];  ///< Internal buffer for IPersist.

    protected:
        // Fields of all simple types for testing.
        Smp::Bool     m_bool;       ///< Internal field of type Bool.
        Smp::Char8    m_char8;      ///< Internal field of type Char8.
        Smp::Int8     m_int8;       ///< Internal field of type Int8.
        Smp::Int16    m_int16;      ///< Internal field of type Int16.
        Smp::Int32    m_int32;      ///< Internal field of type Int32.
        Smp::Int64    m_int64;      ///< Internal field of type Int64.
        Smp::UInt8    m_uint8;      ///< Internal field of type UInt8.
        Smp::UInt16   m_uint16;     ///< Internal field of type UInt16.
        Smp::UInt32   m_uint32;     ///< Internal field of type UInt32.
        Smp::UInt64   m_uint64;     ///< Internal field of type UInt64.
        Smp::Float32  m_float32;    ///< Internal field of type Float32.
        Smp::Float64  m_float64;    ///< Internal field of type Float64.
        Smp::DateTime m_datetime;   ///< Internal field of type DateTime.
        Smp::Duration m_duration;   ///< Internal field of type Duration.
```

```
        // Access to the simulation services
        Smp::Services::ILogger*     m_logger;       ///< Logger service
        Smp::Services::IScheduler*  m_scheduler;    ///< Scheduler service
        Smp::Services::ITimeKeeper* m_timeKeeper;   ///< Time Keeper service

        /// Entry Point implementation.
        virtual void _MyEntryPoint();

        /// Event Sink implementation.
        /// @param sender Sender of event.
        /// @param args Event arguments.
        virtual void _MyEventSink(Smp::IObject* sender, Smp::Float64 args);

    public:
        Smp::IReference*   MyReference;     ///< Example of a Reference.
        Smp::IContainer*   MyContainer;     ///< Example of a Container.
        Smp::IEntryPoint*  MyEntryPoint;    ///< Example of an EntryPoint.
        Smp::IEventSink*   MyEventSink;     ///< Example of an EventSink.
        Smp::IEventSource* MyEventSource;   ///< Example of an EventSource.

        // ------------------------------------------------------------------
        // ----------------------------- IModel -----------------------------
        // ------------------------------------------------------------------

        /// Request for publication.
        void Publish(Smp::IPublication *receiver) throw (Smp::IModel::InvalidModelState);

        /// Connect model to simulator.
        void Connect(Smp::ISimulator *simulator) throw (Smp::IModel::InvalidModelState);

        // ------------------------------------------------------------------
        // ---------------------------- IPersist ----------------------------
        // ------------------------------------------------------------------

        /// Restore model state from storage.
        /// @param    reader Interface that allows reading from storage.
        void Restore(Smp::IStorageReader* reader) throw (CannotRestore);

        /// Store model state to storage.
        /// @param    writer Interface that allows writing to storage.
        void Store(Smp::IStorageWriter* writer) throw (CannotStore);

        // ------------------------------------------------------------------
        // ---------------------------- IDynamicInvocation  -----------------
        // ------------------------------------------------------------------

        /// Create request.
        Smp::IRequest* CreateRequest(Smp::String8 operationName);

        /// Dynamic invocation of operation.
        void Invoke(Smp::IRequest* request) throw (
            Smp::IDynamicInvocation::InvalidOperationName,
            Smp::IDynamicInvocation::InvalidParameterCount,
            Smp::IDynamicInvocation::InvalidParameterType);

        /// Delete request.
        void DeleteRequest(Smp::IRequest* request);
    };
}

#endif  // EXAMPLES_H_
```

Please note that this definition does not include any of the Mdk header files. Therefore, a model based on this class can be distributed without any explicit dependency on the Mdk (e.g. by distributing a DLL with the binary model, together with this header file of its definition).

4.1.2 Implementation of Sample

Both the internal implementation of the entry point and the internal implementation of the event sink are kept very simple, as they only serve as placeholders for a real implementation.

```
/// Entry Point implementation.
```

```
void Examples::Sample::_MyEntryPoint()
{
    // As an example, emit an event to my event source
    (dynamic_cast<MyEventTypeSource*>(MyEventSource))->Emit(this, 1.5);
}

/// Event Sink implementation.
/// @param sender Sender of event.
/// @param args Event arguments.
void Examples::Sample::_MyEventSink(Smp::IObject* sender, Smp::Float64 args)
{
    std::cout
        << "Event Sink called from "
        << sender->GetName()
        << " with value "
        << args
        << std::endl;
}
```

4.1.3 Implementation of IModel

The `Publish()` method publishes the protected fields. For most primitive types, an overloaded method exists which only needs the address of the field to publish. Only for `DataTime` and `Duration` (which are mapped to `Int64`), the type has to be specified as well using its Uuid.

```
/// Request for publication.
void Examples::Sample::Publish(Smp::IPublication* receiver) throw
(Smp::IModel::InvalidModelState)
{
    assert(receiver != NULL);

    // Here, all fields can be published
    receiver->PublishField("Char8",    "Char8 field",    &m_char8);
    receiver->PublishField("Bool",     "Bool field",     &m_bool);
    receiver->PublishField("Int8",     "Int8 field",     &m_int8);
    receiver->PublishField("Int16",    "Int16 field",    &m_int16);
    receiver->PublishField("Int32",    "Int32 field",    &m_int32);
    receiver->PublishField("Int64",    "Int64 field",    &m_int64);
    receiver->PublishField("UInt8",    "UInt8 field",    &m_uint8);
    receiver->PublishField("UInt16",   "UInt16 field",   &m_uint16);
    receiver->PublishField("UInt32",   "UInt32 field",   &m_uint32);
    receiver->PublishField("UInt64",   "UInt64 field",   &m_uint64);
    receiver->PublishField("Float32",  "Float32 field",  &m_float32);
    receiver->PublishField("Float64",  "Float64 field",  &m_float64);

    // These two types need the additional type parameter,
    // as these types map to the same type as Int64 in C++.
    receiver->PublishField("DateTime", "DateTime field", (void*) &m_datetime,
                                                         Smp::Publication::Uuid_DateTime);
    receiver->PublishField("Duration", "Duration field", (void*) &m_duration,
                                                         Smp::Publication::Uuid_Duration);
}

/// Connect model to simulator.
void Examples::Sample::Connect(Smp::ISimulator* simulator) throw
(Smp::IModel::InvalidModelState)
{
    using namespace Smp::Services;

    assert(simulator != NULL);

    // Get access to the logger service
    m_logger     = simulator->GetLogger();

    assert(m_logger != NULL);

    // Get access to the time keeper service
    m_timeKeeper = simulator->GetTimeKeeper();

    assert(m_timeKeeper != NULL);

    // Get access to the time keeper service
```

```
    m_scheduler = simulator->GetScheduler();

    assert(m_scheduler != NULL);

    // Schedule my entry point
    m_scheduler->AddSimulationTimeEvent (
        MyEntryPoint,                           // Entry Point to call
        Smp::Mdk::Duration("01:15:00.5").ticks, // First scheduled at
        Smp::Mdk::Duration(1.5).ticks,          // Seconds between calls
        2);                                     // Number of repeats

    // Write EpochTime to Logger
    Smp::Mdk::DateTime epoch = m_timeKeeper->GetEpochTime();
    Smp::Char8 buffer[BUFFER_SIZE];

    epoch.Get(buffer);

    m_logger->Log(this, buffer, Smp::Services::LMK_Information);
}

/// Request for initialization.
void Examples::Sample::Initialize()
{
}
```

4.1.4 Implementation of IPersist

This example demonstrates how to store and restore a memory block. Note that exception handling is done, which is very important to ensure that the simulation environment can perform `Store()` and `Restore()`.

```
/// Restore model state from storage.
/// @param    reader Interface that allows reading from storage.
void Examples::Sample::Restore(Smp::IStorageReader* reader) throw (
    Smp::IPersist::CannotRestore)
{
    assert(reader != NULL);

    try
    {
        reader->Restore(&m_buffer, sizeof(m_buffer));
    }
    catch (...)
    {
        throw Smp::IPersist::CannotRestore("Error reading from reader");
    }
}

/// Store model state to storage.
/// @param    writer Interface that allows writing to storage.
void Examples::Sample::Store(Smp::IStorageWriter* writer) throw (
    Smp::IPersist::CannotStore)
{
    assert(writer != NULL);

    try
    {
        writer->Store(&m_buffer, sizeof(m_buffer));
    }
    catch (...)
    {
        throw Smp::IPersist::CannotStore("Error writing to writer");
    }
}
```

4.1.5 Implementation of IDynamicInvocation

In this implementation of dynamic invocation, only access to the entry point is provided. As this entry point has neither parameters nor a return type, it is very easy to provide a request object for it. For any other operation name, a null reference is returned, but there is no exception thrown.

When calling the entry point via dynamic invocation, its name and parameter count are validated.

```
/// Create request.
Smp::IRequest* Examples::Sample::CreateRequest(Smp::String8 operationName)
{
    if (strcmp(operationName, EntryPointName) == 0)
    {
        return new Smp::Mdk::Request(EntryPointName, Smp::ST_None);
    }
    else
    {
        return NULL;
    }
}

/// Dynamic invocation of operation.
void Examples::Sample::Invoke(Smp::IRequest* request) throw (
    Smp::IDynamicInvocation::InvalidOperationName,
    Smp::IDynamicInvocation::InvalidParameterCount,
    Smp::IDynamicInvocation::InvalidParameterType)
{
    if (request)
    {
        if (strcmp(request->GetOperationName(), EntryPointName) == 0)
        {
            if (request->GetParameterCount() == 0)
            {
                _MyEntryPoint();
            }
            else
            {
                throw Smp::IDynamicInvocation::InvalidParameterCount(
                    request->GetOperationName(),
                    0,
                    request->GetParameterCount());
            }
        }
        else
        {
            throw Smp::IDynamicInvocation::InvalidOperationName(
                request->GetOperationName());
        }
    }
}

/// Delete request.
void Examples::Sample::DeleteRequest(Smp::IRequest* request)
{
    assert(request != NULL);

    if (request != NULL)
    {
        delete request;
    }
}
```

4.2 An Unmanaged Example using only Helper Classes

The Sample model is now fully implemented making it an unmanaged model. This is first done without deriving from any Mdk base classes. Therefore, this first example provides an explicit implementation for all of the interfaces.

1. The model provides an explicit implementation of `IObject` and `IComponent`. For that, the class declares private fields for name, description and parent.

2. The model provides an explicit implementation of `IAggregate` and `IComposite`. For that, the class declares private fields for references and containers.

3. The private fields are initialised in `_Initialize()`, and cleaned up in `_Cleanup()`.

4. As the model is not managed, it needs to provide a constructor with name, description and parent. As the name may be invalid, this constructor may throw an exception.

5. Every model should always provide a virtual destructor.

4.2.1 Definition of SmpSample

```
#ifndef SMP_SAMPLE_H_
#define SMP_SAMPLE_H_

// ----------------------------------------------------------------------------
// ---------------------------- Include files ---------------------------------
// ----------------------------------------------------------------------------

#include "Examples.h"

// ----------------------------------------------------------------------------
// ---------------------------- class SmpSample -------------------------------
// ----------------------------------------------------------------------------

namespace Examples
{
    /// Smp Example Model.
    /// This model does not use the Mdk for its definition, but only
    /// for its implementation. It is not a managed model.
    class SmpSample : public virtual Sample
    {
    private:
        char* m_name;                   ///< Name given to instance.
        char* m_description;            ///< Description of instance.
        Smp::IComposite*    m_parent;   ///< Parent component.
        Smp::ModelStateKind m_state;    ///< Current model state.
        Smp::ReferenceCollection*   m_references;
        Smp::ContainerCollection*   m_containers;

        void _Initialise();             ///< Initialise internal fields.
        void _Cleanup();                ///< Clean up internal fields.

    public:
        // ------------------------------------------------------------------
        // ---------------------------- Constructor -------------------------
        // ------------------------------------------------------------------

        /// Constructor setting name, description and parent.
        SmpSample(
            Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent) throw (Smp::InvalidObjectName);

        /// Virtual destructor that is called by inherited classes as well.
        virtual ~SmpSample();

        // ------------------------------------------------------------------
```

```
        // ---------------------------- IObject ------------------------------
        // -------------------------------------------------------------------

        /// Returns the name of the object ("property getter").
        Smp::String8 GetName() const;

        /// Returns the description of the object ("property getter").
        Smp::String8 GetDescription() const;

        // -------------------------------------------------------------------
        // ---------------------------- IComponent ---------------------------
        // -------------------------------------------------------------------

        /// Returns the parent component of the component ("property getter").
        Smp::IComposite* GetParent() const;

        // -------------------------------------------------------------------
        // ---------------------------- IModel -------------------------------
        // -------------------------------------------------------------------

        /// Return the state the model is currently in.
        Smp::ModelStateKind GetState() const;

        /// Request for publication.
        void Publish(Smp::IPublication* receiver)
                                    throw (Smp::IModel::InvalidModelState);

        /// Request for configuration.
        void Configure(Smp::Services::ILogger* logger)
                                    throw (Smp::IModel::InvalidModelState);

        /// Connect model to simulator.
        void Connect(Smp::ISimulator* simulator)
                                    throw (Smp::IModel::InvalidModelState);

        // -------------------------------------------------------------------
        // ---------------------------- IAggregate ---------------------------
        // -------------------------------------------------------------------

        /// Get all references.
        const Smp::ReferenceCollection* GetReferences() const;

        /// Get a reference by name.
        Smp::IReference* GetReference(Smp::String8 name) const;

        // -------------------------------------------------------------------
        // ---------------------------- IComposite ---------------------------
        // -------------------------------------------------------------------

        /// Get all containers.
        const Smp::ContainerCollection* GetContainers() const;

        /// Get a container by name.
        Smp::IContainer* GetContainer(Smp::String8 name) const;
    };
}

#endif  // SMP_SAMPLE_H_
```

4.2.2 Implementation of SmpSample

The model first needs to include some Mdk header files.

```
// ---------------------------------------------------------------------------
// ---------------------------- Include files --------------------------------
// ---------------------------------------------------------------------------

#include <iostream>

#include "SmpSample.h"
#include "Mdk/Reference.h"
#include "Mdk/Container.h"
#include "Mdk/EntryPoint.h"
```

```
#include "Mdk/EventSink.h"
#include "Mdk/EventSource.h"
```

4.2.2.1 Initialisation of SmpSample

The Mdk template classes are used in the `_Initialise()` method. Each of the five interface instances is implemented with a single call to one of the template classes provided by the Mdk. As the references and containers have to be returned as a collection on request, they are pushed into the private fields.

For the entry point as well as for the event sink, the instance pointer (`this`) has to be turned into a pointer of type `Sample` first. This is because `_MyEntryPoint()` and `_MyEventSink()` are defined in the Sample base class, not in the derived class `SmpSample`.

To make the code more readable, the two types `MyEventTypeSource` and `MyEventTypeSink` are defined.

```
// ----------------------------------------------------------------------------
// ---------------------------- Typedefs --------------------------------------
// ----------------------------------------------------------------------------

// These two type definitions are not part of the C++ mapping, as formally, an
// event type is not mapped to a C++ type. For convenience, these two types are
// defined based on the Mdk templates

typedef Smp::Mdk::EventSource<Smp::Float64> MyEventTypeSource;
typedef Smp::Mdk::EventSink<Smp::Float64>   MyEventTypeSink;

// ----------------------------------------------------------------------------
// ---------------------------- Implementation --------------------------------
// ----------------------------------------------------------------------------

/// Initialise private fields.
void Examples::SmpSample::_Initialise()
{
    // Setup Aggregation using one reference that accepts all models
    MyReference = new Smp::Mdk::Reference<Smp::IModel>(ReferenceName, "Reference", this);

    m_references = new Smp::ReferenceCollection();
    m_references->push_back(MyReference);

    // Setup Composition using one container that accepts all models
    MyContainer = new Smp::Mdk::Container<Smp::IModel>(ContainerName, "Container", this);

    m_containers = new Smp::ContainerCollection();
    m_containers->push_back(MyContainer);

    // Setup EntryPoints for internal method _MyEntryPoint
    MyEntryPoint = new Smp::Mdk::EntryPoint(EntryPointName, "Sample entry point",
                                            (Sample*) this, &SmpSample::_MyEntryPoint);

    // Setup Event Source
    MyEventSource = new MyEventTypeSource(EventSourceName, "Sample event source", this);

    // Setup Event Sink for internal event handler _MyEventSink
    MyEventSink   = new MyEventTypeSink(EventSinkName, "Sample event sink",
                                            (Sample*) this, &SmpSample::_MyEventSink);
}
```

4.2.2.2 Cleanup of SmpSample

All local fields are released on termination.

```
// Release private fields.
void Examples::SmpSample::_Cleanup()
{
    if (MyReference   != NULL) { delete MyReference;   }
    if (MyContainer   != NULL) { delete MyContainer;   }
    if (MyEntryPoint  != NULL) { delete MyEntryPoint;  }
    if (MyEventSource != NULL) { delete MyEventSource; }
```

```
    if (MyEventSink   != NULL) { delete MyEventSink;   }

    if (m_containers)
    {
        m_containers->clear();
        delete m_containers;
    }

    if (m_references)
    {
        m_references->clear();
        delete m_references;
    }
}
```

4.2.2.3 Constructor of SmpSample

The constructor first initialises the private fields. Then, it checks for a valid name, and stores name, description and parent in its private fields.

```
// ----------------------------------------------------------------------------
// ---------------------------- Constructor -----------------------------------
// ----------------------------------------------------------------------------

/// Constructor setting name, description and parent.
Examples::SmpSample::SmpSample(Smp::String8 name,
                               Smp::String8 description,
                               Smp::IComposite* parent) throw (
                               Smp::InvalidObjectName)
    : m_name(NULL),
      m_description(NULL),
      m_parent(parent),
      m_state(Smp::MSK_Created)
{
    // The name must not be undefined, and must not be empty
    assert(name != NULL);

    // Initialise private fields
    _Initialise();

    // Set name
    if (Smp::Mdk::Object::ValidateName(name))
    {
        m_name = new char[strlen(name) + 1];
        strcpy(m_name, name);
    }
    else
    {
        throw Smp::InvalidObjectName(name);
    }

    // Set description
    if (description != NULL)
    {
        m_description = new char[strlen(description) + 1];
        strcpy(m_description, description);
    }
}
```

4.2.2.4 Destructor of SmpSample

The destructor releases the private fields. Most of the fields (except for name and description) are released by the `Cleanup()` method.

```
/// Virtual destructor that is called by inherited classes as well.
Examples::SmpSample::~SmpSample()
{
    // Release name
    if (m_name != NULL)
    {
        delete[] m_name;
```

```
    }

    // Release description
    if (m_description != NULL)
    {
        delete[] m_description;
    }

    // Clean-up private fields
    _Cleanup();
}
```

4.2.3 Implementation of IObject

These methods are easily implemented without making use of the Mdk.

```
// ----------------------------------------------------------------------------
// ---------------------------- IObject ---------------------------------------
// ----------------------------------------------------------------------------

/// Returns the name of the object ("property getter").
Smp::String8 Examples::SmpSample::GetName() const
{
    return m_name;
}

/// Returns the description of the object ("property getter").
Smp::String8 Examples::SmpSample::GetDescription() const
{
    return m_description;
}
```

4.2.4 Implementation of IComponent

These methods are easily implemented without making use of the Mdk.

```
// ----------------------------------------------------------------------------
// ---------------------------- IComponent ------------------------------------
// ----------------------------------------------------------------------------

/// Returns the parent component of the component ("property getter").
Smp::IComposite* Examples::SmpSample::GetParent() const
{
    return m_parent;
}
```

4.2.5 Implementation of IAggregate

These methods are easily implemented without making use of the Mdk.

```
// ----------------------------------------------------------------------------
// ---------------------------- IAggregate ------------------------------------
// ----------------------------------------------------------------------------

/// Get all references.
const Smp::ReferenceCollection* Examples::SmpSample::GetReferences() const
{
    return m_references;
}

/// Get a reference by name.
Smp::IReference* Examples::SmpSample::GetReference(Smp::String8 name) const
{
    if (strcmp(name, ReferenceName) == 0)
    {
        return MyReference;
    }
    else
    {
        return NULL;
```

```
    }
}
```

4.2.6 Implementation of IModel

These methods are easily implemented without making use of the Mdk. They mainly check that the model is in the proper state, call the base implementation in the Sample class, and change the model state.

```
// ------------------------------------------------------------------------
// --------------------------------- IModel -------------------------------
// ------------------------------------------------------------------------

/// Return the state the model is currently in.
Smp::ModelStateKind Examples::SmpSample::GetState() const
{
    return m_state;
}

/// Request for publication.
void Examples::SmpSample::Publish(Smp::IPublication* receiver) throw
(Smp::IModel::InvalidModelState)
{
    if (m_state == Smp::MSK_Created)
    {
        // Perform state transition before starting publication.
        m_state = Smp::MSK_Publishing;

        // Publish fields using base class
        Sample::Publish(receiver);
    }
    else
    {
        throw Smp::IModel::InvalidModelState(m_state, Smp::MSK_Created);
    }
}

/// Request for configuration.
void Examples::SmpSample::Configure(Smp::Services::ILogger*) throw
(Smp::IModel::InvalidModelState)
{
    if (m_state == Smp::MSK_Publishing)
    {
        // Perform state transition after finishing publication.
        m_state = Smp::MSK_Configured;
    }
    else
    {
        throw Smp::IModel::InvalidModelState(m_state, Smp::MSK_Publishing);
    }
}

/// Connect model to simulator.
void Examples::SmpSample::Connect(Smp::ISimulator* simulator) throw
(Smp::IModel::InvalidModelState)
{
    if (m_state == Smp::MSK_Configured)
    {
        // Perform state transition.
        m_state = Smp::MSK_Connected;

        // Call base class implementation
        Sample::Connect(simulator);
    }
    else
    {
        throw Smp::IModel::InvalidModelState(m_state, Smp::MSK_Configured);
    }
}
```

4.2.7 Implementation of IComposite

These methods are easily implemented without making use of the Mdk.

```
// ----------------------------------------------------------------------------
// --------------------------- IComposite -------------------------------------
// ----------------------------------------------------------------------------

/// Get all containers.
const Smp::ContainerCollection* Examples::SmpSample::GetContainers() const
{
    return m_containers;
}

/// Get a container by name.
Smp::IContainer* Examples::SmpSample::GetContainer(Smp::String8 name) const
{
    if (strcmp(name, ContainerName) == 0)
    {
        return MyContainer;
    }
    else
    {
        return NULL;
    }
}
```

4.3 An Unmanaged Example using Mdk Base Classes

The Sample model is again fully implemented making it an unmanaged model. This is now done deriving from Mdk base classes. Therefore, this second example inherits the implementation for most of the interfaces.

1. The model inherits the implementation of `IObject` and `IComponent`.

2. The model inherits the implementation of `IAggregate` and `IComposite`.

3. The private fields are initialised in `_Initialise()`, and cleaned up in `_Cleanup()`.

4. As the model is not managed, it needs to provide a constructor with name, description and parent. As the name may be invalid, this constructor may throw an exception.

5. Every model should always provide a virtual destructor.

4.3.1 Definition of MdkSample

```
#ifndef MDK_SAMPLE_H_
#define MDK_SAMPLE_H_

// ----------------------------------------------------------------------------
// --------------------------- Include files ----------------------------------
// ----------------------------------------------------------------------------

#include "Examples.h"
#include "Mdk/Model.h"
#include "Mdk/Composite.h"
#include "Mdk/Aggregate.h"

// ----------------------------------------------------------------------------
// --------------------------- class MdkSample --------------------------------
// ----------------------------------------------------------------------------

namespace Examples
{
    /// Mdk Example Model.
    class MdkSample :
```

```
        public virtual Sample,
        public virtual Smp::Mdk::Model,
        public virtual Smp::Mdk::Aggregate,
        public virtual Smp::Mdk::Composite
    {
    private:
        void _Initialise();         ///< Initialise internal fields.
        void _Cleanup();            ///< clean up internal fields.

    public:
        // ------------------------------------------------------------------
        // ---------------------------- Constructor -------------------------
        // ------------------------------------------------------------------

        /// Constructor setting name, description and parent.
        MdkSample(
            Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent) throw (Smp::InvalidObjectName);

        /// Virtual destructor that is called by inherited classes as well.
        virtual ~MdkSample();

        // ------------------------------------------------------------------
        // ---------------------------- IModel ------------------------------
        // ------------------------------------------------------------------

        /// Request for publication.
        void Publish(Smp::IPublication *receiver) throw (Smp::IModel::InvalidModelState);

        /// Connect model to simulator.
        void Connect(Smp::ISimulator *simulator)  throw (Smp::IModel::InvalidModelState);
    };
}

#endif  // MDK_SAMPLE_H_
```

4.3.2 Implementation of MdkSample

The model first needs to include some Mdk header files.

```
// ----------------------------------------------------------------------------
// ---------------------------- Include files ---------------------------------
// ----------------------------------------------------------------------------

#include <iostream>

#include "MdkSample.h"
#include "Mdk/Reference.h"
#include "Mdk/Container.h"
#include "Mdk/EntryPoint.h"
#include "Mdk/EventSink.h"
#include "Mdk/EventSource.h"
```

4.3.2.1 Initialisation of MdkSample

The Mdk template classes are used in the `_Initialise()` method. To make the code more readable, the two types `MyEventTypeSource` and `MyEventTypeSink` are defined. Each of the five interface instances is implemented with a single call to one of the template classes provided by the Mdk.

The reference and container are pushed into a collection using one of the provided convenience methods.

```
// ----------------------------------------------------------------------------
// ---------------------------- Typedefs --------------------------------------
// ----------------------------------------------------------------------------

// These two type definitions are not part of the C++ mapping, as formally, an
// event type is not mapped to a C++ type. For convenience, these two types are
// defined based on the Mdk templates
```



```
typedef Smp::Mdk::EventSource<Smp::Float64> MyEventTypeSource;
typedef Smp::Mdk::EventSink<Smp::Float64>   MyEventTypeSink;

// ------------------------------------------------------------------------
// --------------------------------- Implementation ---------------------------
// ------------------------------------------------------------------------

/// Initialise private fields.
void Examples::MdkSample::_Initialise()
{
    // Setup Aggregation using one reference that accepts all models
    MyReference = new Smp::Mdk::Reference<Smp::IModel>(ReferenceName, "Sample reference",
                                                       this);

    // Use existing implementation to manage References
    this->AddReference(MyReference);

    // Setup Composition using one container that accepts all models
    MyContainer = new Smp::Mdk::Container<Smp::IModel>(ContainerName, "Sample container",
                                                       this);

    // Use existing implementation to manage Containers
    this->AddContainer(MyContainer);

    // Setup EntryPoints for internal method _MyEntryPoint
    MyEntryPoint = new Smp::Mdk::EntryPoint(EntryPointName, "Sample entry point",
                                            (Sample*) this, &MdkSample::_MyEntryPoint);

    // Setup Event Source
    MyEventSource = new MyEventTypeSource(EventSourceName, "Sample event source", this);

    // Setup Event Sink for internal event handler _MyEventSink
    MyEventSink   = new MyEventTypeSink(EventSinkName, "Sample event sink",
                                        (Sample*) this, &MdkSample::_MyEventSink);
}
```

4.3.2.2 Cleanup of MdkSample

All local fields are released on termination.

```
// Release private fields.
void Examples::MdkSample::_Cleanup()
{
    if (MyReference   != NULL) { delete MyReference;   }
    if (MyContainer   != NULL) { delete MyContainer;   }
    if (MyEntryPoint  != NULL) { delete MyEntryPoint;  }
    if (MyEventSource != NULL) { delete MyEventSource; }
    if (MyEventSink   != NULL) { delete MyEventSink;   }
}
```

4.3.2.3 Constructor of MdkSample

The constructor initialises the private fields. It uses the implementation of the base class to set name, description and parent.

```
// ------------------------------------------------------------------------
// --------------------------------- Constructor ------------------------------
// ------------------------------------------------------------------------

/// Constructor setting name, description and parent.
Examples::MdkSample::MdkSample(Smp::String8 name,
                               Smp::String8 description,
                               Smp::IComposite* parent) throw (
                               Smp::InvalidObjectName)
                               : Model(name, description, parent)
{
    // Initialise private fields
    _Initialise();
}
```

4.3.2.4 Destructor of MdkSample

The destructor releases the private fields. All fields are released by the `Cleanup()` method, or by the base classes.

```
/// Virtual destructor that is called by inherited classes as well.
Examples::MdkSample::~MdkSample()
{
    // Clean-up private fields
    _Cleanup();
}
```

4.3.3 Implementation of IModel

The methods are delegated to the base class, but first call the `Mdk::Model` methods, to pass receiver and simulator to the base class.

```
// ------------------------------------------------------------------
// ---------------------------- IModel ------------------------------
// ------------------------------------------------------------------

/// Request for publication.
void Examples::MdkSample::Publish(Smp::IPublication *receiver) throw
(Smp::IModel::InvalidModelState)
{
    // Call Mdk implementation first
    Smp::Mdk::Model::Publish(receiver);

    // Call base class implementation
    Examples::Sample::Publish(receiver);
}

/// Connect model to simulator.
void Examples::MdkSample::Connect(Smp::ISimulator *simulator) throw
(Smp::IModel::InvalidModelState)
{
    // Call Mdk implementation first
    Smp::Mdk::Model::Connect(simulator);

    // Call base class implementation
    Examples::Sample::Connect(simulator);
}
```

4.4 A Managed Example using only Helper Classes

The Sample model is now fully implemented making it a managed model. This is first done without deriving from any Mdk base classes. Therefore, this third example provides an explicit implementation for all of the managed interfaces.

1. The model provides an explicit implementation of `IManagedObject`, `IManagedComponent` and `IManagedModel`. For that, the class declares private fields for name, description and parent.

2. The model provides an explicit implementation of `IAggregate` and `IComposite`, with a managed container (`IManagedContainer`) and reference (`IManagedReference`). For that, the class declares private fields for references and containers.

3. The model provides an explicit implementation of `IEntryPointPublisher`. For that, the class declares a private field for entry points.

4. The model provides an explicit implementation of `IEventProvider`. For that, the class declares a private field for event sources.

5. The model provides an explicit implementation of `IEventConsumer`. For that, the class declares a private field for event sinks.

6. The private fields are initialised in `_Initialise()`, and cleaned up in `_Cleanup()`.

7. As the model is managed, it needs to provide a default constructor. In addition, it provides a constructor with name, description and parent. As the name may be invalid, this constructor may throw an exception.

8. Every model should always provide a virtual destructor.

4.4.1 Definition of SmpManagedSample

```
#ifndef SMP_MANAGED_SAMPLE_H_
#define SMP_MANAGED_SAMPLE_H_

// ----------------------------------------------------------------------------
// --------------------------- Include files ----------------------------------
// ----------------------------------------------------------------------------

#include "Examples.h"
#include "Smp/Management/IManagedModel.h"
#include "Smp/Management/IManagedReference.h"
#include "Smp/Management/IManagedContainer.h"
#include "Smp/Management/IEntryPointPublisher.h"
#include "Smp/Management/IEventConsumer.h"
#include "Smp/Management/IEventProvider.h"

// ----------------------------------------------------------------------------
// --------------------------- class SmpManagedSample -------------------------
// ----------------------------------------------------------------------------

namespace Examples
{
    /// Smp Managed Example model.
    class SmpManagedSample :
        public virtual Sample,
        public virtual Smp::Management::IManagedModel,
        public virtual Smp::Management::IEntryPointPublisher,
        public virtual Smp::Management::IEventConsumer,
        public virtual Smp::Management::IEventProvider
    {
    private:
        char* m_name;                   ///< Name given to instance
        char* m_description;            ///< Description of instance
        Smp::IComposite* m_parent;      ///< Parent component
        Smp::ModelStateKind m_state;    ///< Current model state.
        Smp::ReferenceCollection*   m_references;
        Smp::ContainerCollection*   m_containers;
        Smp::EntryPointCollection*  m_entryPoints;
        Smp::EventSourceCollection* m_eventSources;
        Smp::EventSinkCollection*   m_eventSinks;

        void _Initialise();             ///< Initialise internal fields.
        void _Cleanup();                ///< clean up internal fields.

    public:
        /// Example of a Managed Reference
        Smp::Management::IManagedReference*   MyManagedReference;
        /// Example of a Managed Container
        Smp::Management::IManagedContainer*   MyManagedContainer;

    public:
        // ------------------------------------------------------------------
        // ---------------------------- Constructor -------------------------
        // ------------------------------------------------------------------

        /// Default constructor.
        SmpManagedSample();

        /// Constructor setting name, description and parent.
        SmpManagedSample(
            Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent) throw (Smp::InvalidObjectName);
```

```
        /// Virtual destructor that is called by inherited classes as well.
        virtual ~SmpManagedSample();

        // ------------------------------------------------------------------
        // ---------------------------- IObject -----------------------------
        // ------------------------------------------------------------------

        /// Returns the name of the object ("property getter").
        Smp::String8 GetName() const;

        /// Returns the description of the object ("property getter").
        Smp::String8 GetDescription() const;

        // ------------------------------------------------------------------
        // --------------------------- IComponent ---------------------------
        // ------------------------------------------------------------------

        /// Returns the parent component of the component ("property getter").
        Smp::IComposite* GetParent() const;

        // ---------------------------- IModel ------------------------------
        // ------------------------------------------------------------------

        /// Return the state the model is currently in.
        Smp::ModelStateKind GetState() const;

        /// Request for publication.
        void Publish(Smp::IPublication* receiver)
                                        throw (Smp::IModel::InvalidModelState);

        /// Request for configuration.
        void Configure(Smp::Services::ILogger* logger)
                                        throw (Smp::IModel::InvalidModelState);

        /// Connect model to simulator.
        void Connect(Smp::ISimulator* simulator)
                                        throw (Smp::IModel::InvalidModelState);

        // ------------------------------------------------------------------
        // --------------------------- IAggregate ---------------------------
        // ------------------------------------------------------------------

        /// Get all references.
        const Smp::ReferenceCollection* GetReferences() const;

        /// Get a reference by name.
        Smp::IReference* GetReference(Smp::String8 name) const;

        // ------------------------------------------------------------------
        // --------------------------- IComposite ---------------------------
        // ------------------------------------------------------------------

        /// Get all containers.
        const Smp::ContainerCollection* GetContainers() const;

        /// Get a container by name.
        Smp::IContainer* GetContainer(Smp::String8 name) const;

        // ------------------------------------------------------------------
        // --------------------------- IManagedObject -----------------------
        // ------------------------------------------------------------------

        /// Defines the name of the managed object ("property setter").
        void SetName(Smp::String8 name) throw (Smp::InvalidObjectName);

        /// Defines the description of the managed object ("property setter").
        void SetDescription(Smp::String8 description);

        // ------------------------------------------------------------------
        // --------------------------- IManagedComponent --------------------
        // ------------------------------------------------------------------

        /// Defines the parent component ("property setter").
        void SetParent(Smp::IComposite* parent);
```

```
        // ----------------------------------------------------------------------
        // ---------------------------- IManagedModel ---------------------------
        // ----------------------------------------------------------------------

        /// Get the value of a field which is typed by a system type.
        Smp::AnySimple GetFieldValue(Smp::String8 fullName) throw (
            Smp::Management::IManagedModel::InvalidFieldName);

        /// Set the value of a field which is typed by a system type.
        void SetFieldValue(
            Smp::String8 fullName,
            const Smp::AnySimple value) throw (
            Smp::Management::IManagedModel::InvalidFieldName,
            Smp::Management::IManagedModel::InvalidFieldValue);

        /// Get the value of an array field which is typed by a system type.
        void GetArrayValue(
            Smp::String8 fullName,
            const Smp::AnySimpleArray values,
            const Smp::Int32 length) throw (
            Smp::Management::IManagedModel::InvalidFieldName,
            Smp::Management::IManagedModel::InvalidArraySize);

        /// Set the value of an array field which is typed by a system type.
        void SetArrayValue(
            Smp::String8 fullName,
            const Smp::AnySimpleArray values,
            const Smp::Int32 length) throw (
            Smp::Management::IManagedModel::InvalidFieldName,
            Smp::Management::IManagedModel::InvalidArraySize,
            Smp::Management::IManagedModel::InvalidArrayValue);

        // ----------------------------------------------------------------------
        // ---------------------------- IEntryPointPublisher ------------------
        // ----------------------------------------------------------------------

        /// Get all entry points.
        const Smp::EntryPointCollection* GetEntryPoints() const;

        /// Get an entry point by name.
        const Smp::IEntryPoint* GetEntryPoint(Smp::String8 name) const;

        // ----------------------------------------------------------------------
        // ---------------------------- IEventConsumer ------------------------
        // ----------------------------------------------------------------------

        /// Get all event sources.
        const Smp::EventSourceCollection *GetEventSources() const;

        /// Get an event source by name.
        Smp::IEventSource *GetEventSource(Smp::String8 name) const;

        // ----------------------------------------------------------------------
        // ---------------------------- IEventProvider ------------------------
        // ----------------------------------------------------------------------

        /// Get all event sinks.
        const Smp::EventSinkCollection* GetEventSinks() const;

        /// Get an event sink by name.
        Smp::IEventSink* GetEventSink(Smp::String8 name) const;
    };
}
```

4.4.2 Implementation of SmpManagedSample

The model first needs to include some Mdk helper classes.

```
// ----------------------------------------------------------------------
// ---------------------------- Include files ---------------------------
// ----------------------------------------------------------------------
```

```
#include <iostream>

#include "SmpManagedSample.h"
#include "Mdk/Reference.h"
#include "Mdk/Container.h"
#include "Mdk/EntryPoint.h"
#include "Mdk/EventSink.h"
#include "Mdk/EventSource.h"
#include "Mdk/Management/ManagedReference.h"
#include "Mdk/Management/ManagedContainer.h"
```

4.4.2.1 Initialisation of SmpManagedSample

The Mdk template classes are used in the `_Initialise()` method. To make the code more readable, the two types `MyEventTypeSource` and `MyEventTypeSink` are defined. Each of the five interface instances is implemented with a single call to one of the template classes provided by the Mdk. This time, both the reference and the container are managed, so managed template classes are used. These have additional parameters for the lower and upper bounds.

Each of the five instances is pushed into a collection.

```
// ----------------------------------------------------------------------------
// ---------------------------- Typedefs --------------------------------------
// ----------------------------------------------------------------------------

// These two type definitions are not part of the C++ mapping, as formally, an
// event type is not mapped to a C++ type. For convenience, these two types are
// defined based on the Mdk templates

typedef Smp::Mdk::EventSource<Smp::Float64> MyEventTypeSource;
typedef Smp::Mdk::EventSink<Smp::Float64>   MyEventTypeSink;

// ----------------------------------------------------------------------------
// ---------------------------- Implementation --------------------------------
// ----------------------------------------------------------------------------

/// Initialise private fields
void Examples::SmpManagedSample::_Initialise()
{
    // Setup Aggregation
    MyManagedReference = new Smp::Mdk::Management::ManagedReference<Smp::IModel>(
                         ReferenceName, "Sample reference", this, 0, -1);
    MyReference = MyManagedReference;

    m_references = new Smp::ReferenceCollection();
    m_references->push_back(MyReference);

    // Setup Composition
    MyManagedContainer = new Smp::Mdk::Management::ManagedContainer<Smp::IModel>(
                         ContainerName, "Sample container", this, 0, -1);
    MyContainer = MyManagedContainer;

    m_containers = new Smp::ContainerCollection();
    m_containers->push_back(MyContainer);

    // Setup EntryPoints
    MyEntryPoint = new Smp::Mdk::EntryPoint(EntryPointName, "Sample entry point",
                                    (Sample*) this, &SmpManagedSample::_MyEntryPoint);

    m_entryPoints  = new Smp::EntryPointCollection();
    m_entryPoints->push_back(MyEntryPoint);

    // Setup Event Sources
    MyEventSource = new MyEventTypeSource(EventSourceName, "Sample event source", this);

    m_eventSources = new Smp::EventSourceCollection();
    m_eventSources->push_back(MyEventSource);

    // Setup Event Sinks
    MyEventSink   = new MyEventTypeSink(EventSinkName, "Sample event sink",
                                    (Sample*) this, &SmpManagedSample::_MyEventSink);
```

```
    m_eventSinks    = new Smp::EventSinkCollection();
    m_eventSinks->push_back(MyEventSink);
}
```

4.4.2.2 Cleanup of SmpManagedSample

All local fields are released on termination.

```
// Release private fields.
void Examples::SmpManagedSample::_Cleanup()
{
    if (MyReference    != NULL) { delete MyReference;    }
    if (MyContainer    != NULL) { delete MyContainer;    }
    if (MyEntryPoint   != NULL) { delete MyEntryPoint;   }
    if (MyEventSource  != NULL) { delete MyEventSource;  }
    if (MyEventSink    != NULL) { delete MyEventSink;    }

    if (m_containers)
    {
        m_containers->clear();
        delete m_containers;
    }

    if (m_references)
    {
        m_references->clear();
        delete m_references;
    }

    if (m_entryPoints)
    {
        m_entryPoints->clear();
        delete m_entryPoints;
    }

    if (m_eventSources)
    {
        m_eventSources->clear();
        delete m_eventSources;
    }

    if (m_eventSinks)
    {
        m_eventSinks->clear();
        delete m_eventSinks;
    }
}
```

4.4.2.3 Constructors of SmpManagedSample

Two constructors are provided, which first initialises the private fields. The constructor with arguments uses the methods `SetName()` and `SetDescription()` to set name and description.

```
// ----------------------------------------------------------------------------
// ---------------------------- Constructors ----------------------------------
// ----------------------------------------------------------------------------

/// Default constructor.
Examples::SmpManagedSample::SmpManagedSample()
    : m_name(NULL), m_description(NULL), m_parent(NULL), m_state(Smp::MSK_Created)
{
    _Initialise();
}

/// Constructor setting name, description and parent.
Examples::SmpManagedSample::SmpManagedSample(Smp::String8 name,
                                             Smp::String8 description,
                                             Smp::IComposite* parent) throw (
                                             Smp::InvalidObjectName)
    : m_name(NULL), m_description(NULL), m_parent(parent), m_state(Smp::MSK_Created)
{
```

```
    _Initialise();

    SetName(name);
    SetDescription(description);
}
```

4.4.2.4 Destructor of SmpManagedSample

The destructor releases the private fields. Most of the fields (except for name and description) are released by the `Cleanup()` method.

```
/// Virtual destructor that is called by inherited classes as well.
Examples::SmpManagedSample::~SmpManagedSample()
{
    // Release name
    if (m_name != NULL)
    {
        delete[] m_name;
    }

    if (m_description != NULL)
    {
        delete[] m_description;
    }

    // Clean-up private fields
    _Cleanup();
}
```

4.4.3 Implementation of IObject

These methods are easily implemented without making use of the Mdk.

```
// ----------------------------------------------------------------------------
// ---------------------------- IObject ---------------------------------------
// ----------------------------------------------------------------------------

/// Returns the name of the object ("property getter").
Smp::String8 Examples::SmpManagedSample::GetName() const
{
    return m_name;
}

/// Returns the description of the object ("property getter").
Smp::String8 Examples::SmpManagedSample::GetDescription() const
{
    return m_description;
}
```

4.4.4 Implementation of IComponent

These methods are easily implemented without making use of the Mdk.

```
// ----------------------------------------------------------------------------
// ---------------------------- IComponent ------------------------------------
// ----------------------------------------------------------------------------

/// Returns the parent component of the component ("property getter").
Smp::IComposite* Examples::SmpManagedSample::GetParent() const
{
    return m_parent;
}
```

4.4.5 Implementation of IModel

These methods are easily implemented without making use of the Mdk. They mainly check that the model is in the proper state, call the base implementation in the Sample class, and change the model state.

```
// ----------------------------------------------------------------------------
// -------------------------------- IModel ------------------------------------
// ----------------------------------------------------------------------------

/// Return the state the model is currently in.
{
    return m_state;
}

/// Request for publication.
void Examples::SmpManagedSample::Publish(Smp::IPublication* receiver) throw
(Smp::IModel::InvalidModelState)
{
    if (m_state == Smp::MSK_Created)
    {
        // Perform state transition before starting publication.
        m_state = Smp::MSK_Publishing;

        // Publish fields using base class
        Sample::Publish(receiver);
    }
    else
    {
        throw Smp::IModel::InvalidModelState(m_state, Smp::MSK_Created);
    }
}

/// Request for configuration.
void Examples::SmpManagedSample::Configure(Smp::Services::ILogger* logger) throw
(Smp::IModel::InvalidModelState)
{
    if (m_state == Smp::MSK_Publishing)
    {
        // Perform state transition after finishing publication.
        m_state = Smp::MSK_Configured;
    }
    else
    {
        throw Smp::IModel::InvalidModelState(m_state, Smp::MSK_Publishing);
    }
}

/// Connect model to simulator.
void Examples::SmpManagedSample::Connect(Smp::ISimulator* simulator) throw
(Smp::IModel::InvalidModelState)
{
    if (m_state == Smp::MSK_Configured)
    {
        // Perform state transition.
        m_state = Smp::MSK_Connected;

        // Call base class implementation
        Sample::Connect(simulator);
    }
    else
    {
        throw Smp::IModel::InvalidModelState(m_state, Smp::MSK_Configured);
    }
}
```

4.4.6 Implementation of IAggregate

These methods are easily implemented without making use of the Mdk.

```
// ----------------------------------------------------------------------------
// ---------------------------- IAggregate ------------------------------------
// ----------------------------------------------------------------------------

/// Get all references.
const Smp::ReferenceCollection* Examples::SmpManagedSample::GetReferences() const
{
    return m_references;
}
```

```
/// Get a reference by name.
Smp::IReference* Examples::SmpManagedSample::GetReference(Smp::String8 name) const
{
    if (strcmp(name, ReferenceName) == 0)
    {
        return MyReference;
    }
    else
    {
        return NULL;
    }
}
```

4.4.7 Implementation of IComposite

These methods are easily implemented without making use of the Mdk.

```
// ----------------------------------------------------------------------------
// ---------------------------- IComposite ------------------------------------
// ----------------------------------------------------------------------------

/// Get all containers.
const Smp::ContainerCollection* Examples::SmpManagedSample::GetContainers() const
{
    return m_containers;
}

/// Get a container by name.
Smp::IContainer* Examples::SmpManagedSample::GetContainer(Smp::String8 name) const
{
    if (strcmp(name, ContainerName) == 0)
    {
        return MyContainer;
    }
    else
    {
        return NULL;
    }
}
```

4.4.8 Implementation of IManagedObject

The `SetName()` method makes use of the static `ValidateName()` method of `Smp::Mdk::Object`.

```
// ----------------------------------------------------------------------------
// ---------------------------- IManagedObject --------------------------------
// ----------------------------------------------------------------------------

/// Defines the name of the managed object ("property setter").
void Examples::SmpManagedSample::SetName(Smp::String8 name) throw (
    Smp::InvalidObjectName)
{
    // Check given name using static method of Mdk Object class
    if (Smp::Mdk::Object::ValidateName(name))
    {
        // Delete old name only when new name is valid
        if (m_name != NULL)
        {
            delete[] m_name;
        }

        m_name = new char[strlen(name) + 1];
        strcpy(m_name, name);
    }
    else
    {
        throw Smp::InvalidObjectName(name);
    }
}

/// Defines the description of the managed object ("property setter").
```

```
/// Management components may use this to set object descriptions.
/// @param   description Description of object.
void Examples::SmpManagedSample::SetDescription(Smp::String8 description)
{
    // Delete existing description
    if (m_description != NULL)
    {
        delete[] m_description;
    }

    // Check for NULL pointer
    if (description)
    {
        m_description = new char[strlen(description) + 1];
        strcpy(m_description, description);
    }
}
```

4.4.9 Implementation of IManagedComponent

The `SetParent()` method is easily implemented without making use of the Mdk.

```
// ----------------------------------------------------------------------------
// ---------------------------- IManagedComponent -----------------------------
// ----------------------------------------------------------------------------

/// Defines the parent component ("property setter").
void Examples::SmpManagedSample::SetParent(Smp::IComposite* parent)
{
    m_parent = parent;
}
```

4.4.10 Implementation of IManagedModel

The methods `GetFieldValue()` and `SetFieldValue()` should work for all fields of the model. To keep the implementation short, only two of the 14 published fields are handled here.

```
// ----------------------------------------------------------------------------
// ---------------------------- IManagedModel ---------------------------------
// ----------------------------------------------------------------------------

/// Get the value of a field which is typed by a system type.
Smp::AnySimple Examples::SmpManagedSample::GetFieldValue(Smp::String8 fullName) throw (
    Smp::Management::IManagedModel::InvalidFieldName)
{
    Smp::Mdk::AnySimple fieldValue;
    fieldValue.type = Smp::ST_None;

    if (strcmp(fullName, "Bool") == 0)
    {
        fieldValue.Set(m_bool);
    }
    else if (strcmp(fullName, "Char8") == 0)
    {
        fieldValue.Set(m_char8);
    }
    else // Here are missing the other 12 published fields
    {
        throw Smp::Management::IManagedModel::InvalidFieldName(fullName);
    }

    return fieldValue;
}

/// Set the value of a field which is typed by a system type.
void Examples::SmpManagedSample::SetFieldValue(
    Smp::String8 fullName,
    const Smp::AnySimple value) throw (
    Smp::Management::IManagedModel::InvalidFieldName,
    Smp::Management::IManagedModel::InvalidFieldValue)
{
```

```
    if (strcmp(fullName, "Bool") == 0)
    {
        if (value.type == Smp::ST_Bool)
        {
            m_bool = value.value.boolValue;
        }
        else
        {
            throw Smp::InvalidAnyType(value.type, Smp::ST_Bool);
        }
    }
    else if (strcmp(fullName, "Char8") == 0)
    {
        if (value.type == Smp::ST_Char8)
        {
            m_char8 = value.value.char8Value;
        }
        else
        {
            throw Smp::InvalidAnyType(value.type, Smp::ST_Char8);
        }
    }
    else // Here are missing the other 12 published fields
    {
        throw Smp::Management::IManagedModel::InvalidFieldName(fullName);
    }
}

/// Get the value of an array field which is typed by a system type.
void Examples::SmpManagedSample::GetArrayValue(
    Smp::String8 fullName,
    const Smp::AnySimpleArray values,
    const Smp::Int32 length) throw (
    Smp::Management::IManagedModel::InvalidFieldName,
    Smp::Management::IManagedModel::InvalidArraySize)
{
    // This model does not have array fields
    throw Smp::Management::IManagedModel::InvalidFieldName(fullName);
}

/// Set the value of an array field which is typed by a system type.
void Examples::SmpManagedSample::SetArrayValue(
    Smp::String8 fullName,
    const Smp::AnySimpleArray values,
    const Smp::Int32 length) throw (
    Smp::Management::IManagedModel::InvalidFieldName,
    Smp::Management::IManagedModel::InvalidArraySize,
    Smp::Management::IManagedModel::InvalidArrayValue)
{
    // This model does not have array fields
    throw Smp::Management::IManagedModel::InvalidFieldName(fullName);
}
```

4.4.11 Implementation of IEntryPointPublisher

These methods are easily implemented without making use of the Mdk.

```
// ----------------------------------------------------------------------------
// --------------------------- IEntryPointPublisher ---------------------------
// ----------------------------------------------------------------------------

/// Get all entry points.
const Smp::EntryPointCollection* Examples::SmpManagedSample::GetEntryPoints() const
{
    return m_entryPoints;
}

/// Get an entry point by name.
const Smp::IEntryPoint* Examples::SmpManagedSample::GetEntryPoint(Smp::String8 name)
const
{
    if (strcmp(name, Examples::EntryPointName) == 0)
    {
```

```
        return MyEntryPoint;
    }
    else
    {
        return NULL;
    }
}
```

4.4.12 Implementation of IEventConsumer

These methods are easily implemented without making use of the Mdk.

```
// ------------------------------------------------------------------------
// ---------------------------- IEventConsumer ----------------------------
// ------------------------------------------------------------------------

/// Get all event sources.
const Smp::EventSourceCollection *Examples::SmpManagedSample::GetEventSources() const
{
    return m_eventSources;
}

/// Get an event source by name.
Smp::IEventSource *Examples::SmpManagedSample::GetEventSource(Smp::String8 name) const
{
    if (strcmp(name, Examples::EventSourceName) == 0)
    {
        return MyEventSource;
    }
    else
    {
        return NULL;
    }
}
```

4.4.13 Implementation of IEventProvider

These methods are easily implemented without making use of the Mdk.

```
// ------------------------------------------------------------------------
// ---------------------------- IEventProvider ----------------------------
// ------------------------------------------------------------------------

/// Get all event sinks.
const Smp::EventSinkCollection* Examples::SmpManagedSample::GetEventSinks() const
{
    return m_eventSinks;
}

/// Get an event sink by name.
Smp::IEventSink* Examples::SmpManagedSample::GetEventSink(Smp::String8 name) const
{
    if (strcmp(name, Examples::EventSinkName) == 0)
    {
        return MyEventSink;
    }
    else
    {
        return NULL;
    }
}
```

4.5 A Managed Example using Mdk Base Classes

The Sample model is again fully implemented making it a managed model. This is now done deriving from Mdk base classes. Therefore, this last example inherits the implementation for most of the managed interfaces from Mdk base classes.

1. The model inherits the implementation of `IManagedObject` and `IManagedComponent`.

2. The model delegates the implementation of `IManagedModel` to `IPublication`.

3. The model inherits the implementation of `IAggregate` and `IComposite`.

4. The model inherits the implementation of `IEntryPointPublisher`.

5. The model inherits the implementation of `IEventProvider`.

6. The model inherits the implementation of `IEventConsumer`.

7. The private fields are initialised in `_Initialise()`, and cleaned up in `_Cleanup()`.

8. As the model is managed, it needs to provide a default constructor. In addition, it provides a constructor with name, description and parent. As the name may be invalid, this constructor may throw an exception.

9. Every model should always provide a virtual destructor.

4.5.1 Definition of MdkManagedSample

```
#ifndef MDK_MANAGED_SAMPLE_H_
#define MDK_MANAGED_SAMPLE_H_

// ----------------------------------------------------------------------------
// ---------------------------- Include files ---------------------------------
// ----------------------------------------------------------------------------

#include "Examples.h"
#include "Smp/Management/IManagedReference.h"
#include "Smp/Management/IManagedContainer.h"
#include "Mdk/Composite.h"
#include "Mdk/Aggregate.h"
#include "Mdk/Management/ManagedModel.h"
#include "Mdk/Management/EntryPointPublisher.h"
#include "Mdk/Management/EventConsumer.h"
#include "Mdk/Management/EventProvider.h"

// ----------------------------------------------------------------------------
// ---------------------------- class SmpManagedSample ------------------------
// ----------------------------------------------------------------------------

namespace Examples
{
    /// Mdk Managed Example Model.
    class MdkManagedSample :
        public virtual Sample,
        public virtual Smp::Mdk::Aggregate,
        public virtual Smp::Mdk::Composite,
        public virtual Smp::Mdk::Management::ManagedModel,
        public virtual Smp::Mdk::Management::EntryPointPublisher,
        public virtual Smp::Mdk::Management::EventConsumer,
        public virtual Smp::Mdk::Management::EventProvider
    {
    private:
        void _Initialise();         ///< Initialise internal fields.
        void _Cleanup();            ///< clean up internal fields.

    public:
        /// Example of a Managed Reference
        Smp::Management::IManagedReference*   MyManagedReference;
        /// Example of a Managed Container
        Smp::Management::IManagedContainer*   MyManagedContainer;

    public:
        // ------------------------------------------------------------------
        // ---------------------------- Constructor -------------------------
        // ------------------------------------------------------------------
```

```
        /// Default constructor.
        MdkManagedSample();

        /// Constructor setting name, description and parent.
        MdkManagedSample(
            Smp::String8 name,
            Smp::String8 description,
            Smp::IComposite* parent) throw (Smp::InvalidObjectName);

        /// Virtual destructor that is called by inherited classes as well.
        virtual ~MdkManagedSample();

        // ------------------------------------------------------------------
        // ----------------------------- IModel -----------------------------
        // ------------------------------------------------------------------

        /// Request for publication.
        void Publish(Smp::IPublication *receiver) throw (Smp::IModel::InvalidModelState);

        /// Connect model to simulator.
        void Connect(Smp::ISimulator *simulator) throw (Smp::IModel::InvalidModelState);
    };
}

#endif  // MDK_MANAGED_SAMPLE_H_
```

4.5.2 Implementation of MdkManagedSample

The model first needs to include some Mdk helper classes.

```
// ----------------------------------------------------------------------------
// ---------------------------- Include files ---------------------------------
// ----------------------------------------------------------------------------

#include <iostream>

#include "MdkManagedSample.h"
#include "Mdk/EntryPoint.h"
#include "Mdk/EventSink.h"
#include "Mdk/EventSource.h"
#include "Mdk/Management/ManagedReference.h"
#include "Mdk/Management/ManagedContainer.h"
```

4.5.2.1 Initialisation of MdkManagedSample

The Mdk template classes are used in the `_Initialize()` method. To make the code more readable, the two types `MyEventTypeSource` and `MyEventTypeSink` are defined. Each of the five interface instances is implemented with a single call to one of the template classes provided by the Mdk. This time, both the reference and the container are managed, so managed template classes are used. These have additional parameters for the lower and upper bounds.

Each of the five instances is pushed into a collection using one of the provided convenience methods.

```
// ----------------------------------------------------------------------------
// ---------------------------- Typedefs --------------------------------------
// ----------------------------------------------------------------------------

// These two type definitions are not part of the C++ mapping, as formally, an
// event type is not mapped to a C++ type. For convenience, these two types are
// defined based on the Mdk templates

typedef Smp::Mdk::EventSource<Smp::Float64> MyEventTypeSource;
typedef Smp::Mdk::EventSink<Smp::Float64>   MyEventTypeSink;

// ----------------------------------------------------------------------------
// ---------------------------- Implementation --------------------------------
// ----------------------------------------------------------------------------

/// Initialise private fields
void Examples::MdkManagedSample::_Initialise()
```

```
{
    // Setup Aggregation
    MyManagedReference = new Smp::Mdk::Management::ManagedReference<Smp::IModel>(
                         ReferenceName, "Sample reference", this, 0, -1);
    MyReference = MyManagedReference;

    // Use existing implementation to manage References
    this->AddReference(MyReference);

    // Setup Composition
    MyManagedContainer = new Smp::Mdk::Management::ManagedContainer<Smp::IModel>(
                         ContainerName, "Sample container", this, 0, -1);
    MyContainer = MyManagedContainer;

    // Use existing implementation to manage Containers
    this->AddContainer(MyContainer);

    // Setup EntryPoints
    MyEntryPoint = new Smp::Mdk::EntryPoint(EntryPointName, "Sample entry point",
                                    (Sample*) this, &MdkManagedSample::_MyEntryPoint);

    // Use existing implementation to manage Entry Points
    this->AddEntryPoint(MyEntryPoint);

    // Setup Event Sources
    MyEventSource = new MyEventTypeSource(EventSourceName, "Sample event source", this);

    // Use existing implementation to manage Event Sources
    this->AddEventSource(MyEventSource);

    // Setup Event Sinks
    MyEventSink   = new MyEventTypeSink(EventSinkName, "Sample event sink",
                                    (Sample*) this, &MdkManagedSample::_MyEventSink);

    // Use existing implementation to manage Event Sinks
    this->AddEventSink(MyEventSink);
}
```

4.5.2.2 Cleanup of MdkManagedSample

All local fields are released on termination.

```
// Release private fields.
void Examples::MdkManagedSample::_Cleanup()
{
    if (MyReference   != NULL) { delete MyReference;   }
    if (MyContainer   != NULL) { delete MyContainer;   }
    if (MyEntryPoint  != NULL) { delete MyEntryPoint;  }
    if (MyEventSource != NULL) { delete MyEventSource; }
    if (MyEventSink   != NULL) { delete MyEventSink;   }
}
```

4.5.2.3 Constructors of MdkManagedSample

Two constructors are provided, which initialise the private fields. The constructor with arguments uses the implementation of the base class to set name, description and parent.

```
// ----------------------------------------------------------------------------
// ---------------------------- Constructors ----------------------------------
// ----------------------------------------------------------------------------

/// Default constructor.
/// Every managed model should have a default constructor. It is needed
/// for creating a class factory.
Examples::MdkManagedSample::MdkManagedSample()
{
    // Initialise private fields
    _Initialise();
}

/// Constructor setting name, description and parent.
/// @param name Name of new instance.
```

```
/// @param description Description of new instance.
/// @param parent Parent of new instance.
Examples::MdkManagedSample::MdkManagedSample(Smp::String8 name,
                                    Smp::String8 description,
                                    Smp::IComposite* parent) throw (
                                    Smp::InvalidObjectName)
    : Smp::Mdk::Management::ManagedModel(name, description, parent)
{
    // Initialise private fields
    _Initialise();
}
```

4.5.2.4 Destructor of MdkManagedSample

The destructor releases the private fields. All fields are released by the `Cleanup()` method, or by the base classes.

```
/// Virtual destructor that is called by inherited classes as well.
Examples::MdkManagedSample::~MdkManagedSample()
{
    // Clean-up private fields
    _Cleanup();
}
```

4.5.3 Implementation of IModel

The methods are delegated to the base class, but first call the `Mdk::Model` methods, to pass receiver and simulator to the base class.

```
// ------------------------------------------------------------------
// --------------------------- IModel -------------------------------
// ------------------------------------------------------------------

/// Request for publication.
void Examples::MdkManagedSample::Publish(Smp::IPublication *receiver)
 throw (Smp::IModel::InvalidModelState)
{
    // Call Mdk implementation first
    Smp::Mdk::Model::Publish(receiver);

    // Call base class implementation
    Examples::Sample::Publish(receiver);
}

/// Connect model to simulator.
void Examples::MdkManagedSample::Connect(Smp::ISimulator *simulator)
 throw (Smp::IModel::InvalidModelState)
{
    // Call Mdk implementation first
    Smp::Mdk::Model::Connect(simulator);

    // Call base class implementation
    Examples::Sample::Connect(simulator);
}
```

This Page is Intentionally left Blank