Chapter 17
Type DynAny

17.1 Chapter Overview

This chapter discusses the DynAny interface and its derived interfaces. The DynAny interface permits you to compose and decompose complex values at run time even without compile-time knowledge of the IDL definitions involved. Section 17.3 presents the IDL and functionality for DynAny and its derived types. Section 17.4 explains how to use DynAny from C++, and Sections 17.5 and 17.6 present a few applications of DynAny.

17.2 Introduction

As you saw in Chapters 15 and 16, to insert a user-defined value into an Any you must have compile-time knowledge of the corresponding IDL type because to insert a value into an Any, you must use the corresponding overloaded <<= operator generated by the IDL compiler.

This inability to construct Any values on-the-fly is a severe drawback for some applications. For example, debuggers, generic user interfaces for objects, and services such as the OMG Notification Service [26] all require the ability to interpret values without knowing the values' IDL types at compile time.

The DynAny interface was added to CORBA with the 2.2 revision to permit applications to dynamically compose and decompose any values. In a nutshell, the DynAny interface does for any values what the TypeCode interface does for type codes. DynAny permits applications to compose a value at run time whose type was unknown when the application was compiled, and to transmit that value as an any. Similarly, DynAny allows applications to receive a value of type any from an operation invocation and both to interpret the type of the any (using the TypeCode interface) and to extract its value (using the DynAny interface) without compile-time knowledge of the IDL types involved.

Unfortunately, the DynAny interfaces published with CORBA 2.2 contained a number of defects. As a result, the interfaces were (incompatibly) revised with CORBA 2.3, which is the version we describe here. If you need to find out which version is supported by your ORB, look for the definition of the DynAny interface. If the definition appears inside the DynamicAny module, you have the 2.3 version; if the definition appears inside the CORBA module, you have the (now obsolete) 2.2 version.

The DynAny interface is large, so we follow the same approach here as in Chapter 16: we first present the IDL interface for DynAny and then illustrate its use in C++ with a few examples.

17.3 The DynAny Interface

The DynAny API is composed of nine interfaces. One of these, interface DynAnyFactory, allows you to create DynAny objects. The other interfaces are DynAny and seven interfaces derived from DynAny, as shown in Figure 17.1.

All these interfaces are defined in the DynamicAny module. The derived interfaces, such as DynFixed and DynStruct, are used to create any values of the corresponding type (DynStruct is used both for structures and for exceptions). The DynAny base interface deals with any values containing other IDL types, such as strings, object references, and so on.[1]

1. Note that DynValue represents an any containing an object-by-value. Because we do not cover OBV in this book, we do not cover DynValue (see [18] for details).

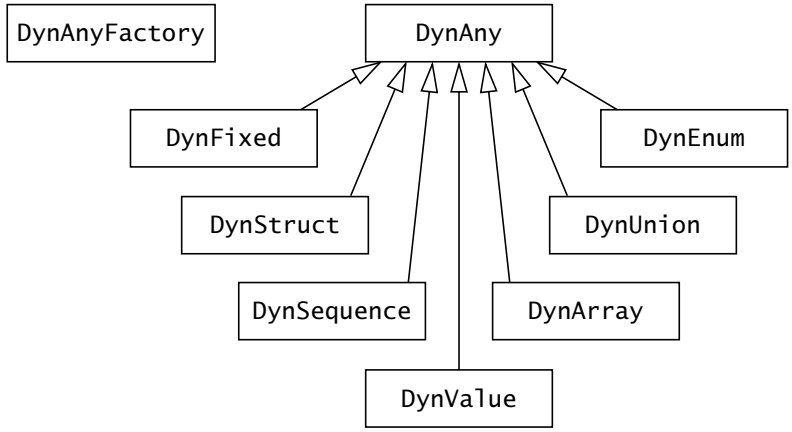

Figure 17.1. Interface inheritance hierarchy for `DynAny`.

17.3.1 Locality Constraints

`DynAny` and `DynAnyFactory` are locality-constrained interfaces. This means that you cannot pass instances of `DynFactory` or `DynAny` and its derived interfaces over the wire, and you cannot stringify references to these interfaces with `ORB::object_to_string`. Otherwise, locality-constrained objects are like ordinary objects. In particular, they implicitly inherit from `Object` and therefore support operations such as `is_a` and `is_equivalent`.

`DynAny` allows you to compose and decompose values of type `any`. To dynamically compose an `any` value and send it across an interface, you first construct a `DynAny` object and then extract the corresponding `any` value from it. Similarly, to dynamically decompose an `any` value, you initialize a `DynAny` object from the `any` value and use the `DynAny` object for decomposition.

17.3.2 IDL for `DynAny`

The IDL for `DynAny` is large, so we present it in stages here. The functionality relating to `DynAny` falls into the following broad categories.

- Creation operations
- Life cycle operations (copying and destroying `DynAny` objects)
- Type code operations (setting and retrieving the type code of `DynAny` objects)

- Insertion operations (inserting values of basic type into DynAny objects to compose complex types)
- Extraction operations (extracting values of basic type from DynAny objects to decompose them)
- Iteration operations (getting from one component of a DynAny to the next)
- Conversion operations between DynAny and any values

DynAny Creation

Before we look at the DynAny interface itself, we must consider how to create a DynAny object. The creation operations for DynAny are provided by the DynAnyFactory interface:

```
module DynamicAny {
    interface DynAny;   // Forward declaration

    interface DynAnyFactory {
        exception InconsistentTypeCode {};

        DynAny create_dyn_any(in any value)
                    raises(InconsistentTypeCode);
        DynAny create_dyn_any_from_type_code(in CORBA::TypeCode t)
                    raises(InconsistentTypeCode);
    };
    // ...
};
```

You obtain a reference to the factory by passing the string "DynAnyFactory" to ORB::resolve_initial_references.

The fundamental creation operation is create_dyn_any, which constructs a DynAny object from an any value. The new DynAny object contains the same type code as the any value passed to the operation.

If the any value passed to create_dyn_any is *not* a structure, exception, sequence, array, union, enumeration, fixed-point type, or object-by-value, the returned object reference is of type DynAny. Otherwise, the actual run-time type of the reference is DynStruct, DynSequence, and so on, depending on the type of value contained in the value parameter passed to create_dyn_any.

To determine the exact type of a DynAny, you can extract its type code and use the TCKind value of the type code to narrow the reference to the appropriate derived type.

The other creation operation, `create_dyn_any_from_type_code`, creates a default-initialized `DynAny` object for the type code passed as the `t` parameter. Default initialization for simple types assigns a default value as follows.

- Boolean values are set to false.
- Numeric (integral and floating-point) values and values of type `octet`, `char`, and `wchar` are set to zero.
- Values of type `string` or `wstring` (whether bounded or unbounded) are set to the empty string.
- Object references are set to nil.
- Values of type `TypeCode` are set to `tk_null`.[2]
- Values of type `any` are set to contain a `tk_null` type code and no value.

For complex types, default initialization assigns a default value as follows.

- Sequence values are set to the empty sequence.
- Fixed-point values are set to zero.
- Enumerated values are set to the first enumerator indicated by the type code.
- Structure and exception members are set (recursively) to their default values.
- Array elements are set (recursively) to their default values.
- For unions, the discriminator is set to indicate the first named member of the union; that member is set (recursively) to its default value.

Whenever you create a `DynAny` object, the type code associated with the `DynAny` object during creation remains with that object for its lifetime. You cannot change the type code of a `DynAny` object later.

The creation operations raise an `InconsistentTypeCode` exception if you attempt to create a `DynAny` object with an illegal or obsolete type code, such as the deprecated `tk_Principal` type code.

`DynAny` Life Cycle, Assignment, Comparison, and Conversion

Here is the first part of the `DynAny` interface:

2. You can create a `DynAny` for an `any` containing a type code as its value. In that case, the `any` contains a type code indicating `tk_TypeCode` and a type code value. For default initialization, that type code value is set to `tk_null`.

```
module DynamicAny {
    // ...
    interface DynAny {
        exception InvalidValue {};
        exception TypeMismatch {};

        // Assignment and life cycle operations
        void    assign(in DynAny dyn_any) raises(TypeMismatch);
        DynAny  copy();
        void    destroy();

        // Comparison
        boolean equal(in DynAny da);

        // Conversion operations
        void    from_any(in any value)
                    raises(TypeMismatch, InvalidValue);
        any     to_any();

        // Type code accessor
        CORBA::TypeCode type();

        // More operations here...
    };
};
```

The life cycle operations `copy` and `destroy` have the usual semantics. The `copy` operation returns a deep copy of a `DynAny`, and the `destroy` operation destroys a `DynAny` (including any `DynAny` objects it may be composed of). Before you release the last reference to a `DynAny` object that was created by one of the factory operations or by the `copy` operation, you must explicitly call `destroy` on the object; otherwise, you may leak memory. Invoking an operation on a destroyed `DynAny` raises `OBJECT_NOT_EXIST`.[3]

The `assign` operation makes a deep assignment of the contents of a `DynAny` object to another `DynAny` object. You can assign `DynAny`s to each other only if both source and target have the same type code (as determined by `TypeCode::equivalent`); otherwise, `assign` raises `TypeMismatch`. The type

3. To the best of our knowledge, all current ORBs do nothing on a call to `destroy` and instead destroy a `DynAny` object when you release its last object reference. However, strictly speaking, the call to `destroy` is required by the specification (even if it does nothing for a particular implementation).

code of a DynAny is set when that DynAny is created and cannot be changed for the lifetime of the DynAny.

The equal operation returns true if the type codes of the two DynAnys are equivalent and if (recursively) all component DynAnys have identical values.

The from_any and to_any operations provide conversion between types any and DynAny. For the from_any operation, you must pass an any with a type code that is equivalent to that of the target DynAny; otherwise, you get a TypeMismatch exception. Similarly, the source any must contain a legal value; for example, passing an any containing a null string raises InvalidValue.

The type operation returns the type code of its DynAny. This operation is useful if you are passed a DynAny for a complex type and you want to narrow that DynAny to a derived type, such as DynSequence.

DynAny Composition

The DynAny interface contains one operation to insert each type of non-structured value into a DynAny. To do this, you must have previously created a DynAny object. The type code of the DynAny must be equivalent to that of the value being inserted; otherwise, the operations raise an InvalidValue exception.

```
interface DynAny {
    // ...

    // Insertion operations
    void    insert_boolean(in boolean value)
                raises(TypeMismatch, InvalidValue);
    void    insert_octet(in octet value)
                raises(TypeMismatch, InvalidValue);
    void    insert_char(in char value)
                raises(TypeMismatch, InvalidValue);
    void    insert_wchar(in wchar value)
                raises(TypeMismatch, InvalidValue);
    void    insert_short(in short value)
                raises(TypeMismatch, InvalidValue);
    void    insert_ushort(in unsigned short value)
                raises(TypeMismatch, InvalidValue);
    void    insert_long(in long value)
                raises(TypeMismatch, InvalidValue);
    void    insert_ulong(in unsigned long value)
                raises(TypeMismatch, InvalidValue);
    void    insert_longlong(in long long value)
                raises(TypeMismatch, InvalidValue);
    void    insert_ulonglong(in unsigned long long value)
                raises(TypeMismatch, InvalidValue);
```

```
    void    insert_float(in float value)
                raises(TypeMismatch, InvalidValue);
    void    insert_double(in double value)
                raises(TypeMismatch, InvalidValue);
    void    insert_longdouble(in long double value)
                raises(TypeMismatch, InvalidValue);
    void    insert_string(in string value)
                raises(TypeMismatch, InvalidValue);
    void    insert_wstring(in wstring value)
                raises(TypeMismatch, InvalidValue);
    void    insert_reference(in Object value)
                raises(TypeMismatch, InvalidValue);
    void    insert_typecode(in CORBA::TypeCode value)
                raises(TypeMismatch, InvalidValue);
    void    insert_any(in any value)
                raises(TypeMismatch, InvalidValue);
    void    insert_dyn_any(in DynAny value)
                raises(TypeMismatch, InvalidValue);
    void    insert_val(in ValueBase value)
                raises(TypeMismatch, InvalidValue);
    // ...
};
```

As you can see, there is one operation for each simple type. Each operation accepts a value and inserts it into a `DynAny`, raising `TypeMismatch` if the value's type does not match that of the operation. The `InvalidValue` exception is raised if the value is unacceptable (such as inserting a string that exceeds the bound of a bounded string). `InvalidValue` is also raised if you attempt to insert a value into a `DynAny` that has components but has a current position of −1 (see page 746).

The `insert_any` operation inserts an `any` value into the `any` represented by the `DynAny`. (The net effect is that one `any` value is nested inside another.)

The `insert_dyn_any` operation does the same thing as `insert_any` but accepts a `DynAny` parameter. This is useful if you have just constructed an `any` value as a `DynAny` and now want to insert it into another `DynAny` (because it saves the need to convert the `DynAny` to an `any` before insertion).

`DynAny` Decomposition

To complement the insertion operations, `DynAny` also contains operations to extract values from a `DynAny`. As with insertion, the operation must match the type code of the `DynAny`; otherwise, it raises a `TypeMismatch` exception. Attempts to extract a value from a `DynAny` that has components, but has a current position of −1, raise `InvalidValue` (see page 746).

```
interface DynAny {
    // ...

    // Extraction operations
    boolean             get_boolean()
                            raises(TypeMismatch, InvalidValue);
    octet               get_octet()
                            raises(TypeMismatch, InvalidValue);
    char                get_char()
                            raises(TypeMismatch, InvalidValue);
    wchar               get_wchar()
                            raises(TypeMismatch, InvalidValue);
    short               get_short()
                            raises(TypeMismatch, InvalidValue);
    unsigned short      get_ushort()
                            raises(TypeMismatch, InvalidValue);
    long                get_long()
                            raises(TypeMismatch, InvalidValue);
    unsigned long       get_ulong()
                            raises(TypeMismatch, InvalidValue);
    long long           get_longlong()
                            raises(TypeMismatch, InvalidValue);
    unsigned long long  get_ulonglong()
                            raises(TypeMismatch, InvalidValue);
    float               get_float()
                            raises(TypeMismatch, InvalidValue);
    double              get_double()
                            raises(TypeMismatch, InvalidValue);
    long double         get_longdouble()
                            raises(TypeMismatch, InvalidValue);
    string              get_string()
                            raises(TypeMismatch, InvalidValue);
    wstring             get_wstring()
                            raises(TypeMismatch, InvalidValue);
    Object              get_reference()
                            raises(TypeMismatch, InvalidValue);
    CORBA::TypeCode     get_typecode()
                            raises(TypeMismatch, InvalidValue);
    any                 get_any()
                            raises(TypeMismatch, InvalidValue);
    DynAny              get_dyn_any()
                            raises(TypeMismatch, InvalidValue);
    ValueBase           get_val()
                            raises(TypeMismatch, InvalidValue);
    // ...
};
```

DynAny Iteration

The DynAny interface provides five operations to iterate over the components of a DynAny. Iteration applies only to structures, exceptions, unions, sequences, arrays, and value types. Here are the relevant IDL definitions:

```
interface DynAny {
    // ...

    // Iteration operations
    unsigned long   component_count();
    DynAny          current_component() raises(TypeMismatch);
    boolean         seek(in long index);
    boolean         next();
    void            rewind();
};
```

A DynAny value consists of a type code and an ordered collection of component DynAny values. For example, a DynAny for a structure having four members contains a collection of four DynAny values, one for each member. The iterator operations permit you to selectively examine the contents of the collection.

Each DynAny value maintains a current position in its collection of components. The current position is indexed from 0 to $n-1$, where n is the number of components. For example, for a four-member structure, the index ranges from 0 to 3. The current position of a DynAny can indicate the "no current component" condition; in that case, the index value is –1.

When a DynAny is created, the initial index is zero if that DynAny has components. For example, creating a DynStruct for a four-member structure sets the index to zero, so the current position initially indicates the first member of the structure. On the other hand, creating a DynAny for a type that cannot have components (such as a long or an empty exception) sets the index to –1.

The component_count operation returns the number of components of a DynAny. For simple types, such as long, and for enumerated and fixed-point types, component_count returns zero. For sequences, the operation returns the number of elements in the sequence; for structures and exceptions, it returns the number of members; for arrays, it returns the number of elements; for unions, it returns 2 if a member is active and 1 otherwise.

The current_component operation returns the DynAny for the component at the current position. The current position is not affected by this call, so successive calls to current_component return the same component. (You must explicitly call next or seek to advance to the next component.) Calling current_component on a DynAny that cannot have components (such as a long or an empty exception)

raises TypeMismatch. Calling current_component on a DynAny that has components, but whose current position is −1, returns a nil reference. You can call the destroy operation on non-nil DynAnys returned by current_component. However, the call will have no effect. Instead, you must call destroy on DynAnys created with create_dyn_any, create_dyn_any_from_type_code, or copy.

The next operation increments the current position and returns true if the new current position denotes a component. Otherwise, if you call next with the current position already at the final component, next returns false and sets the current position to −1. If you call next on a DynAny that does not contain components (such as the DynAny for a string), next returns false and leaves the current position at −1.

The seek operation allows you to explicitly set the current position (a value of zero indicates the first component). The seek operation returns true if the position denoted by index points at an existing component. If index denotes a non-existent position, seek returns false and sets the current position to −1. If you call seek on a DynAny that does not have components, seek returns false and leaves the current position at −1.

The rewind operation is equivalent to calling seek(0).

Note that all the insert_*type* and get_*type* operations on DynAny leave the current position unchanged.

If all this seems a bit abstract right now, don't despair—we show examples of iterating over the components of a DynAny in Section 17.4.3.

17.3.3 IDL for DynEnum

The DynEnum interface manipulates values of enumerated type:

```
interface DynEnum : DynAny {
    string          get_as_string();
    void            set_as_string(in string val)
                        raises(InvalidValue);
    unsigned long   get_as_ulong();
    void            set_as_ulong(in unsigned long val)
                        raises(InvalidValue);
};
```

The get_as_string and set_as_string operations provide access to an enumerated value by its IDL identifier. For example, given the enumeration

```
enum Color { red, green, blue };
```

you can set a DynEnum value to red by calling set_as_string("red"). Note that enumerator names are optional in type codes (see Section 16.3.2). As a result, get_as_string returns an empty string if you construct a DynEnum from an any whose type code does not contain enumerator names. In that case, set_as_string raises InvalidValue, as it does if you pass it a string that is outside the range of the enumerated type. (For example, for the Color enumeration, calling set_as_string("black") raises InvalidValue.)

The get_as_ulong and set_as_ulong operations provide access to the ordinal value of an enumerated value. For example, calling set_as_ulong(1) does the same thing as calling set_as_string("green"). However, set_as_ulong works even if the type code for the enumeration does not contain the enumerator identifiers. Passing a value outside the range of the enumerated type to set_as_ulong raises InvalidValue.

17.3.4 IDL for DynStruct

The DynStruct interface allows us to manipulate structures as well as exceptions.

```
typedef string FieldName;

struct NameValuePair {
    FieldName   id;
    any         value;
};
typedef sequence<NameValuePair> NameValuePairSeq;

struct NameDynAnyPair {
    FieldName   id;
    DynAny      value;
};
typedef sequence<NameDynAnyPair> NameDynAnyPairSeq;

interface DynStruct : DynAny {
    FieldName           current_member_name()
                            raises(TypeMismatch, InvalidValue);
    CORBA::TCKind       current_member_kind()
                            raises(TypeMismatch, InvalidValue);
    NameValuePairSeq    get_members();
    void                set_members(in NameValuePairSeq value)
                            raises(TypeMismatch, InvalidValue);
    NameDynAnyPairSeq   get_members_as_dyn_any();
```

```
    void                set_members_as_dyn_any(
                            in NameDynAnyPairSeq value
                        ) raises(TypeMismatch, InvalidValue);
};
```

The main operations are `get_members` and `set_members`. They allow you to set and get the value of the structure or exception members as a sequence of name–value pairs. Each element in the sequence represents one structure member (so for a four-member structure, the sequence would contain four name–value pairs). Each name–value pair contains the name of the structure member (a string) and its value (of type `any`).

You must ensure that a sequence passed to `set_members` has the correct number of elements (one for each structure member) and contains the structure members in the same order as their IDL definition; otherwise, `set_members` raises `TypeMismatch`. The values inserted must be consistent with the members' type codes; otherwise, `set_members` raises `InvalidValue`.

The `current_member_name` operation returns the name of the member at the current position as established by the iterator operations on the `DynAny` base interface. Note that because member names are optional in type codes, `current_member_name` may return an empty string. If the `DynStruct` represents an empty exception, `current_member_name` raises `TypeMismatch`. If the current position is −1, `current_member_name` raises `InvalidValue`.

The `current_member_kind` operation returns the `TCKind` value for the type code of the current member. The exception semantics are the same as for `current_member_name`.

`get_members_as_dyn_any` and `set_members_as_dyn_any` are analogous to `get_members` and `set_members`, but they operate on sequences of name–`DynAny` pairs. These operations are useful if you are working extensively with `DynStructs` because they avoid the need to convert a constructed `DynAny` into an `any` before it can be used to get or set structure members.

17.3.5 IDL for `DynUnion`

The `DynUnion` interface allows us to manipulate unions.

```
interface DynUnion : DynAny {
    DynAny              get_discriminator();
    void                set_discriminator(in DynAny d)
                            raises(TypeMismatch);
    void                set_to_default_member()
                            raises(TypeMismatch);
```

```
    void                set_to_no_active_member()
                            raises(TypeMismatch);
    boolean             has_no_active_member()
                            raises(TypeMismatch);
    CORBA::TCKind       discriminator_kind();
    DynAny              member() raises(InvalidValue);
    FieldName           member_name();
    CORBA::TCKind       member_kind();
};
```

A `DynUnion` has two valid current positions: 0, which denotes the discriminator, and 1, which denotes the active member. `component_count` for a `DynUnion` is 1 if the discriminator value indicates that no member is active; otherwise, it is 2.

The `get_discriminator` operation returns the discriminator value of the union as a `DynAny`.

The `set_discriminator` operation sets the discriminator value of the union. Attempts to set a discriminator value that disagrees with the type code for the union raise `TypeMismatch`. Setting the discriminator can affect the active member and the current position of the union.

- If the discriminator is set to a value that agrees with the currently active member, that member remains active and the current position is set to 1.
- If the discriminator is set to a value that belongs to a member of the union that is not currently active, the currently active member (if any) is destroyed and the member corresponding to the new discriminator value is initialized to its default value. The current position is set to 1.
- If the discriminator is set to a value that indicates that no member should be active, the currently active member (if any) is destroyed and the current position is set to 0.

The `set_to_default_member` operation sets the discriminator to a value that is consistent with the `default` member of the union and sets the current position to 0. If the union does not have an explicit `default` case, the operation raises `TypeMismatch`.

The `set_to_no_active_member` operation sets the discriminator to a value that does not correspond to any of the union's `case` labels. Calling this operation sets the current position to 0 (and causes `component_count` to return 1). If the union has an explicit `default` case, the operation raises `TypeMismatch`.

The `has_no_active_member` operation returns true if the union's discriminator has a value that does not correspond to an active member. In other words, the operation returns true if the union consists solely of a discriminator because no

member is active. The operation returns false for unions with an explicit `default` label and for unions that exhaust the entire discriminator range for explicit `case` labels.

The `member` operation returns the currently active member as a `DynAny`. You can examine (and change) the value of the active member via that `DynAny`. Note that the returned reference remains valid only for as long as the active member remains active. If you use the returned reference after activating a different member, you receive an `OBJECT_NOT_EXIST` exception. Calling `member` on a union that does not currently have an active member raises `InvalidValue`.

The `discriminator_kind` and `member_kind` operations return the `TCKind` value of the discriminator and member type, respectively. The `member_name` operation allows you to read the name of the active member. Because member names are optional within type codes, this operation may return the empty string.

17.3.6 IDL for `DynSequence`

The `DynSequence` interface allows us to manipulate sequences.

```
typedef sequence<any> AnySeq;
typedef sequence<DynAny> DynAnySeq;

interface DynSequence : DynAny {
    unsigned long   get_length();
    void            set_length(in unsigned long len)
                        raises(InvalidValue);
    AnySeq          get_elements();
    void            set_elements(in AnySeq value)
                        raises(TypeMismatch, InvalidValue);
    DynAnySeq       get_elements_as_dyn_any();
    void            set_elements_as_dyn_any(in DynAnySeq value)
                        raises(TypeMismatch, InvalidValue);
};
```

The `get_length` operation returns the number of elements of the sequence.

The `set_length` operation sets the number of elements of the sequence. If you increase the number of elements, new elements are added at the tail of the sequence and are default-initialized. If the current position of the sequence is valid (not –1), increasing the length of the sequence leaves the current position unaffected. Otherwise, if the current position is –1, it is set to indicate the first of the newly added elements. Increasing the length of a sequence beyond its bound raises `InvalidValue`.

Decreasing the length of a sequence removes elements from the tail of the sequence. The current position is set as follows.

- If the current position is −1, it remains at −1.
- If the length of the sequence is set to zero, the current position is set to −1.
- If the current position indicates an element that was not removed when the sequence was shortened, the current position remains unaffected.
- If the current position indicates an element that was removed when the sequence was shortened, the current position is set to −1.

The `get_elements` operation returns the elements of the sequence as a sequence of `any` values. The `set_elements` operation sets the elements of the sequence according to the parameter `value`. `set_elements` completely replaces the sequence's elements and sets the length of the sequence to the number of elements that are passed. The current position is set to −1 if `set_elements` is called with a zero-length sequence; otherwise, the current position is set to 0. If the type of the sequence elements disagrees with the sequence's type code (either some elements are of the wrong type, or the `value` parameter has more elements than the sequence bound allows), the operation raises `TypeMismatch`.

The `get_elements_as_dyn_any` and `set_elements_as_dyn_any` operations behave like `get_elements` and `set_elements`, but (to avoid unnecessary conversions to `any`) they return and accept sequences of `DynAny` elements.

17.3.7 IDL for `DynArray`

The `DynArray` interface allows us to manipulate arrays.

```
interface DynArray : DynAny {
    AnySeq      get_elements();
    void        set_elements(in AnySeq value)
                    raises(TypeMismatch, InvalidValue);
    DynAnySeq   get_elements_as_dyn_any();
    void        set_elements_as_dyn_any(in DynAnySeq value)
                    raises(TypeMismatch, InvalidValue);
};
```

The `get_elements` and `set_elements` operations work as with sequences. However, because arrays have a fixed number of elements, the element sequences always have as many elements as are specified as the array's dimension. `set_elements` sets the current position to 0. `set_elements` raises a `TypeMismatch` exception if you pass a sequence that contains elements that

disagree with the array's type code. If you pass a sequence that is too long or too short, `set_elements` raises `InvalidValue`.

The `get_elements_as_dyn_any` and `set_elements_as_dyn_any` operations have the same semantics as `get_elements` and `set_elements`, but they return and accept sequences of `DynAny` (to avoid unnecessary conversions to `any`).

Note that you can access the dimension of the array via the `component_count` operation.

17.3.8 IDL for `DynFixed`

The `DynFixed` interface allows us to manipulate `any`s containing fixed-point values.

```
interface DynFixed : DynAny {
    string  get_value();
    boolean set_value(in string val)
                    raises(TypeMismatch, InvalidValue);
};
```

IDL does not offer a generic type that could represent fixed-point types with different numbers of digits and scale. Therefore, `DynFixed` uses a string representation to get and set fixed-point values.

The `get_value` operation returns the value of a `DynFixed` as a string. The syntax is the same as for IDL fixed-point constants, with the trailing `d` or `D` being optional. For example, `get_value` can return `1.3`, `1.3d`, or `1.3D`.

The `set_value` operation sets the value of a `DynFixed` using the same syntax. (Again, a trailing `d` or `D` is optional). If `set_value` is passed a string whose scale exceeds the range of the `DynFixed`, the operation raises `InvalidValue`. If the passed string has invalid syntax, `set_value` raises `TypeMismatch`. `set_value` returns true if the passed value can be represented without loss of precision; otherwise, if the string contains too many fractional digits, extraneous fractional digits are truncated and `set_value` returns false.

17.4 C++ Mapping for `DynAny`

The C++ mapping for `DynAny` and its derived interfaces follows the normal mapping rules, so there are no additional memory management rules or parameter changes to consider. Rather than repeat the full interfaces here in their C++

versions, we show a number of examples of how to use DynAny to compose and decompose values of different types.

17.4.1 Using DynAny with Simple Types

The easiest use of DynAny is with simple types. We can use DynAny both to compose and to decompose values. The following code fragment dynamically creates an Any value containing a long with value 20.[4]

```
// Make a DynAny containing a long with value 20.
//
DynamicAny::DynAny_var da
    = daf->create_dyn_any_from_type_code(CORBA::_tc_long);
da->insert_long(20);

// Turn it into an Any
//
CORBA::Any_var an_any = da->to_any();

// Use an_any...

// Destroy the DynAny.
//
da->destroy();  // da and an_any deallocate
                // when they go out of scope
```

This code first creates a new DynAny by calling create_dyn_any_from_type_code with the type code for long, and then it initializes the DynAny by calling insert_long. Now the DynAny is in a defined state, and the code calls to_any to convert it into an Any that can, for example, be passed across an IDL interface. To get rid of the DynAny, the code calls destroy. Note that the variable da calls CORBA::release when it goes out of scope, so it deallocates the reference to the DynAny object.

The preceding code example is naive in the sense that it uses a DynAny variable to create an Any for a simple value. Strictly speaking, there is no point in doing this because we can always create an Any containing a simple value directly without using DynAny. However, if we want to compose user-defined complex types, we must use dynamic creation; the insert operations for simple

4. Note that all code examples in this chapter assume that a reference to a DynAnyFactory was obtained from resolve_initial_references and is available in the variable daf.

types are provided for consistency and to avoid having to deal with DynAny for complex types but with Any for simple types.

Instead of creating a DynAny object by supplying a type code, we can create it from an Any value. Here is the same code again, but this time the DynAny is created with a call to create_dyn_any.

```
// Make an Any containing the value 20 as a long.
//
CORBA::Any an_any;
an_any <<= (CORBA::Long)20;

// Create a DynAny from the Any.
//
DynamicAny::DynAny_var da = daf->create_dyn_any(an_any);

// Use da...

// Destroy the DynAny again.
//
da->destroy();
```

Again, looking at this, there seems little point in using DynAny for a simple type such as long. However, when user-defined complex types are involved, creating a DynAny from an Any becomes important: if an Any contains a value whose type was unknown at compile time, we construct a DynAny from the Any and then use the DynAny to decompose the value into its components.

The extraction operations on DynAny permit decomposition of simple values, but there is little point in using DynAny for this purpose. By definition, simple values are simple and therefore do not need to be decomposed. Instead, we can use the type code constants and Any values to extract simple values. The extraction functions are provided because they make it easier to extract simple values if they appear as components of a complex value (see Section 17.4.3).

For completeness, here is an example that uses DynAny to extract a long value from an Any.

```
CORBA::Any an_any = ...;    // Get any from somewhere...
DynamicAny::DynAny_var da = daf->create_dyn_any(an_any);
CORBA::TypeCode_var tc = da->type();

switch (tc->kind()) {
case CORBA::tk_long:
    {
        CORBA::Long l = da->get_long();
        cout << "long value is " << l << endl;
```

```
    }
    break;
// Other cases here...
}
da->destroy();  // Clean up
```

17.4.2 Using DynEnum

In discussing the `show_label` function in Section 16.4 on page 713, we encounter a problem. Without compile-time knowledge of the IDL, it is impossible to show the label value for a union that has a discriminator of enumerated type. The `DynAny` functionality allows us to get around this problem.

Here again is the relevant part of the `show_label` function, updated here to use `DynAny` for decomposition of the label value:

```
void
show_label(const CORBA::Any * ap)
{
    CORBA::TypeCode_var tc = ap->type();
    if (tc->kind() == CORBA::tk_octet) {
        cout << "default:" << endl;
    } else {
        cout << "case ";
        switch (tc->kind()) {
        // ...
        case CORBA::tk_enum:
            {
                DynamicAny::DynAny_var da
                    = daf->create_dyn_any_from_type_code(tc);
                DynamicAny::DynEnum_var de
                    = DynamicAny::DynEnum::_narrow(da);
                de->from_any(*ap);
                CORBA::String_var s = de->get_as_string();
                cout << s;
                da->destroy();
            }
            break;
        // ...
        }
        cout << ":" << endl;
    }
}
```

The branch of the `switch` statement for enumerated types creates a `DynEnum` by calling `create_dyn_any_from_type_code` and narrowing the returned

reference. We know that this must succeed because we have already established that the Any being decoded has an enumerated value. The next step is to initialize the DynEnum with the actual value by calling from_any. Now the DynEnum is in a well-defined state, and the code calls get_as_string to print the name of the enumerator before it destroys the original DynAny. You must destroy the value—without the call to destroy, the code would leak the DynAny object.

Following is another version of the same code. Instead of explicitly creating a DynAny object from the type code, it initializes a DynAny from the Any:

```
// ...
case CORBA::tk_enum:
    {
        DynamicAny::DynAny_var da = daf->create_dyn_any(*ap);
        DynamicAny::DynEnum_var de
            = DynamicAny::DynEnum::_narrow(da);
        CORBA::String_var s = de->get_as_string();
        cout << s;
        da->destroy();
    }
    break;
// ...
```

We know from the type code that the Any contains an enumerated value. This means that there is no need to test for a nil return value from the call to _narrow because that call cannot possibly fail except by throwing an exception (for example, in case of memory exhaustion).

We can also use DynEnum to dynamically compose an enumerated value even without knowledge of the IDL. To do this, we first construct a type code for the enumerated type and then compose a DynEnum for the value. The following code example dynamically creates the type code for the SearchCriterion type in the climate control system and then sets a DynEnum value to contain the LOCATION enumerator:

```
// Make a type code for the SearchCriterion type
//
CORBA::EnumMemberSeq members;
members.length(3);
members[0] = CORBA::string_dup("ASSET");
members[1] = CORBA::string_dup("LOCATION");
members[2] = CORBA::string_dup("MODEL");

CORBA::TypeCode_var enum_tc
    = orb->create_enum_tc(
        "IDL:acme.com/CCS/Controller/SearchCriterion:1.0",
```

```
        "SearchCriterion", members
      );

// Make an Any with the value LOCATION
//
DynamicAny::DynAny_var da
    = daf->create_dyn_any_from_type_code(enum_tc);  // Create
DynamicAny::DynEnum_var de
    = DynamicAny::DynEnum::_narrow(da);
de->set_as_string("LOCATION");                      // Set value

CORBA::Any_var an_any = de->to_any();               // Extract Any

// Use an_any...

da->destroy();                                      // Clean up
```

17.4.3 Using `DynStruct`

The `DynStruct` class allows us to compose structures and exceptions. Either you can supply member values as a sequence of name–value pairs and set member values with a single call to `set_members` or `set_members_as_dyn_any`, or you can iterate over the members and set each member individually.

Following is a code fragment that composes a `CCS::Thermostat::BtData` structure using the `set_members_as_dyn_any` function. The IDL for this structure is as follows:

```
#pragma prefix "acme.com"

module CCS {
    // ...
    typedef short           TempType;
    // ...
    interface Thermostat : Thermometer {
        struct BtData {
            TempType    requested;
            TempType    min_permitted;
            TempType    max_permitted;
            string      error_msg;
        };
        // ...
    };
    // ...
};
```

The code first constructs the type code for the `BtData` structure and then creates each element for the member sequence. To correctly preserve aliasing information, the code uses DynAny to construct the members of type `TempType`. (Recall from Section 15.4 that we cannot preserve aliases by inserting a simple type directly into an Any.)

```
// Create an alias for short called "TempType".
//
CORBA::TypeCode_var TempType_tc
    = orb->create_alias_tc(
        "IDL:acme.com/CCS/TempType:1.0",
        "TempType", CORBA::_tc_short
      );

// Create a sequence containing the definitions for the
// four structure members.
//
CORBA::StructMemberSeq  mseq;
mseq.length(4);
mseq[0].name = CORBA::string_dup("requested");
mseq[0].type = TempType_tc;
mseq[1].name = CORBA::string_dup("min_permitted");
mseq[1].type = TempType_tc;
mseq[2].name = CORBA::string_dup("max_permitted");
mseq[2].type = TempType_tc;
mseq[3].name = CORBA::string_dup("error_msg");
mseq[3].type = CORBA::TypeCode::_duplicate(CORBA::_tc_string);

// Create a type code for the BtData structure.
//
CORBA::TypeCode_var BtData_tc
    = orb->create_struct_tc(
        "IDL:acme.com/CCS/Thermostat/BtData:1.0",
        "BtData", mseq
      );

// Create DynAny objects for the structure members.
//
DynamicAny::DynAny_var requested
    = daf->create_dyn_any_from_type_code(TempType_tc);
requested->insert_short(99);

DynamicAny::DynAny_var min_permitted
    = daf->create_dyn_any_from_type_code(TempType_tc);
min_permitted->insert_short(50);
```

```
DynamicAny::DynAny_var max_permitted
    = daf->create_dyn_any_from_type_code(TempType_tc);
max_permitted->insert_short(90);

DynamicAny::DynAny_var error_msg
    = daf->create_dyn_any_from_type_code(CORBA::_tc_string);
error_msg->insert_string("Too hot");

// Create the member sequence.
//
DynamicAny::NameDynAnyPairSeq members;
members.length(4);
members[0].id = CORBA::string_dup("requested");
members[0].value = requested;
members[1].id = CORBA::string_dup("min_permitted");
members[1].value = min_permitted;
members[2].id = CORBA::string_dup("max_permitted");
members[2].value = max_permitted;
members[3].id = CORBA::string_dup("error_msg");
members[3].value = error_msg;

// Now create the DynStruct and initialize it.
//
DynamicAny::DynAny_var da
    = daf->create_dyn_any_from_type_code(BtData_tc);
DynamicAny::DynStruct_var ds
    = DynamicAny::DynStruct::_narrow(da);
ds->set_members_as_dyn_any(members);

// Get the Any out of the DynStruct.
//
CORBA::Any_var btd = ds->to_any();

// Use btd...

// Clean up.
//
da->destroy();
requested->destroy();
max_permitted->destroy();
min_permitted->destroy();
error_msg->destroy();
```

Note that the code takes care to call `destroy` for each `DynAny` it has created.

Instead of calling `set_members_as_dyn_any` to initialize the structure, we can iterate over the members and set them individually. For the `BtData` structure, this approach is considerably easier than the preceding one because there is no need to first construct a `DynAny` for each member:

```
// Create type code for BtData as before...
CORBA::TypeCode_var BtData_tc = ...;

// Create DynStruct and initialize members using iteration.
//
DynamicAny::DynAny_var da
    = daf->create_dyn_any_from_type_code(BtData_tc);
DynamicAny::DynStruct_var ds
    = DynamicAny::DynStruct::_narrow(da);
DynamicAny::DynAny_var member;
member = ds->current_component();
member->insert_short(99);           // Set requested
ds->next();
member = ds->current_component();
member->insert_short(50);           // Set min_permitted
ds->next();
member = ds->current_component();
member->insert_short(90);           // Set max_permitted
ds->next();
member = ds->current_component();
member->insert_string("Too hot");   // Set error_msg

CORBA::Any_var btd = ds->to_any();  // Get the Any

// Use btd...

da->destroy();  // Clean up
```

After calling `current_component`, the code calls `next` to advance the current position to the next member. Note that there is no need to explicitly destroy the `DynAny` objects returned by `current_component`; it is sufficient to destroy only `da` because destroying a `DynAny` also destroys its constituent components.

The preceding code correctly preserves aliasing information for the members. For example, the type code for the `requested` member indicates `CCS::TempType` instead of `short` because the type code for `BtData` contains the aliasing information.

To decompose a structure, either we can call `get_members` to extract the members and then decompose each element of the returned sequence, or we can

iterate over the structure and decompose the members one by one. Following is a code fragment that iterates over the components of a `DynStruct` and hands each component to a `display` helper function:

```
DynamicAny::DynStruct_var ds = ...;
for (CORBA::ULong i = 0; i < ds->component_count(); i++) {
    DynamicAny::DynAny_var cc = ds->current_component();
    CORBA::String_var name = ds->current_member_name();
    cout << name << " = ";
    display(cc);
    ds->next();
}
```

This code calls `component_count` to get the number of members and uses that number to control the loop. On each iteration, a call to `next` advances the current position to the next member.

17.4.4 Using `DynUnion`

To compose a union, you must set the discriminator and active member. Following is a code fragment that creates a `KeyType` union for the climate control system:

```
// Create DynUnion.
//
DynamicAny::DynAny_var da
    = daf->create_dyn_any_from_type_code(
        CCS::Controller::_tc_KeyType
      );
DynamicAny::DynUnion_var du = DynamicAny::DynUnion::_narrow(da);

// Set discriminator to LOCATION.
//
DynamicAny::DynAny_var tmp = du->get_discriminator();
DynamicAny::DynEnum_var disc = DynamicAny::DynEnum::_narrow(tmp);
disc->set_as_ulong(1);  // LOCATION

// Set member for LOCATION.
//
DynamicAny::DynAny_var member = du->member();
member->insert_string("Room 414");

// Use du...

da->destroy();  // Clean up
```

For simplicity, the code creates the DynUnion using the generated _tc_KeyType constant, but it could have used a synthesized type code instead.

The first step is to get the DynAny for the discriminator and to narrow that DynAny to a DynEnum interface. This narrowing step must succeed because we know that the union has an enumerated discriminator. The second step sets the discriminator value to indicate that the location member is active. Now that the correct union member is indicated by the discriminator, the code calls the member function on the DynUnion to get the DynAny for the active member and then sets the active member's value using the DynAny returned by member. Finally, the code calls destroy to avoid leaking the DynUnion created initially.

To compose a union that does not have an active member, you use set_to_no_active_member. To compose a union that activates the default member, you can either call set_to_default_member (if you don't care about the precise value of the discriminator) or set the discriminator to a value that activates the default member.

Decomposition of unions follows the general pattern of ensuring that a union member is active, followed by decomposition of that member:

```
DynamicAny::DynUnion_var du = ...;  // Get DynUnion...

DynamicAny::DynAny_var disc = du->get_discriminator();
// Decompose discriminator...

if (!du->has_no_active_member()) {
    CORBA::String_var mname = du->member_name();
    cout << "member name is " << mname << endl;
    DynamicAny::DynAny_var member = du->member();
    // Decompose member...
}
```

17.4.5 Using DynSequence

Composition of sequences presents you with two options. Either you can iterate over the sequence using the DynAny base interface iterator operations, or you can use set_elements or set_elements_as_dyn_any to supply the sequence elements as a sequence of any or DynAny values.

The following code fragment fills a sequence of values using iteration. We assume that the IDL contains a definition LongSeq for a sequence of long values.

```
DynamicAny::DynAny_var da
    = daf->create_dyn_any_from_type_code(_tc_LongSeq);
DynamicAny::DynSequence_var ds
    = DynamicAny::DynSequence::_narrow(da);

ds->set_length(20);
for (CORBA::ULong i = 0; i < ds->component_count(); i++) {
    DynamicAny::DynAny_var elmt = ds->current_component();
    elmt->insert_long(i);
    ds->next();
}

// Use ds...

da->destroy();  // Clean up
```

For decomposition of a sequence, you can either iterate over the individual members or call `get_elements` or `get_elements_as_dyn_any`. Following is a code fragment that extracts the elements from a sequence of long values using `get_elements`. Note that `get_elements` returns a sequence of `Any` (not `DynAny`), so the code extracts the long values from the members for printing:

```
DynamicAny::DynSequence_var ds = ...;

DynamicAny::AnySeq_var as = ds->get_elements();
for (CORBA::ULong i = 0; i < as->length(); i++) {
    CORBA::ULong val;
    as[i] >>= val;
    cout << val << endl;
}
```

17.5 Using DynAny for Generic Display

One useful application of `DynAny` is for generic display purposes. Using `DynAny`, we can decompose an arbitrary `Any` value into its constituent parts at run time and display them on screen. This capability is useful, for example, for debuggers, which must be able to inspect a value even if the value's type was not known at compile time.

Following is an outline for such a generic display function. We have left it incomplete to save space, so not all possible types are dealt with. However, there is enough for you to see how you would complete the function to handle the

remaining types. Note that our display function simply writes to standard output and does not make any attempt to improve the layout of the data. Of course, there is nothing to prevent you from using more-sophisticated means to present the contents of a value, such as list widgets for a graphical user interface.

```
void
display(DynamicAny::DynAny_ptr da)
{
    // Strip aliases
    //
    CORBA::TypeCode_var tc(da->type());
    while (tc->kind() == CORBA::tk_alias)
        tc = tc->content_type();

    // Deal with each type of data.
    //
    switch (tc->kind()) {
    case CORBA::tk_short:
        cout << da->get_short();
        break;
    case CORBA::tk_long:
        cout << da->get_long();
        break;
    case CORBA::tk_string:
        {
            CORBA::String_var s(da->get_string());
            cout << "\"" << s << "\"";
        }
        break;

    // Deal with remaining simple types here... (not shown)
    //
    case CORBA::tk_struct:
    case CORBA::tk_except:
        {
           DynamicAny::DynStruct_var ds =
               DynamicAny::DynStruct::_narrow(da);
           for (int i = 0; i < ds->component_count(); i++) {
               DynamicAny::DynAny_var cm(ds->current_component());
               CORBA::String_var mem(ds->current_member_name());
               cout << mem << " = " << endl;
               display(cm);
               ds->next();
           }
        }
```

```
            break;
        case CORBA::tk_enum:
            {
                DynamicAny::DynEnum_var de
                    = DynamicAny::DynEnum::_narrow(da);
                CORBA::String_var val(de->get_as_string());
                cout << val << endl;
            }
            break;
        case CORBA::tk_objref:
            {
                CORBA::TypeCode_var tc(da->type());
                CORBA::String_var id(tc->id());
                cout << "Object reference (" << id << ")" << endl;
                CORBA::Object_var obj(da->get_reference());
                CORBA::String_var str_ref(orb->object_to_string(obj));
                cout << str_ref << endl;
            }
            break;
        case CORBA::tk_array:
            {
               for (int i = 0; i < da->component_count(); i++) {
                   DynamicAny::DynAny_var cm(da->current_component());
                   cout << "[" << i << "] = " << endl;
                   display(cm);
                   da->next();
               }
            }
            break;

        // Deal with remaining complex types here... (not shown)
        //
        }

        cout << endl;
    }
```

17.6 Obtaining Type Information

When you look at the preceding sections, you will notice that the sample code we have presented still contains type information. However, instead of this type information being in the form of IDL-generated stubs, it is now in the form of manifest constants in the source code, such as literal repository IDs. This means

that the source code still has compile-time knowledge of the IDL types, at least for composition of types. The question really is this: How does an application otherwise (without linking against the stubs and without using manifest constants) obtain the necessary type information to compose values?

The answer depends on the application. For decomposition of values, no compile-time knowledge of the IDL types is required at all. The `TypeCode` and `DynAny` interfaces provide all the necessary functionality to decompose a complex value into its constituent values without any compile-time knowledge of the IDL types. However, for *composition* of values, we clearly need to get type knowledge from somewhere. The following sections present options for getting that type knowledge at run time.

17.6.1 Type Information from the OMG Interface Repository

One option is to consult an interface repository at run time. We do not cover the OMG Interface Repository in this book, so we do not present this option in detail. Suffice it to say that the Interface Repository (IFR) allows you to discover the complete IDL definition of a type at run time by using the type's repository ID as an index into the Interface Repository. The IFR returns object references to type descriptions that fully describe a type. This is similar in nature (if not in detail) to the way type codes describe the type of a value. The main difference between type codes and the IFR is that the IFR can describe things other than value types, such as interfaces, operations, attributes, and modules.

Using the IFR, `DynAny`, and the DII in combination, we can, for example, build a universal CORBA client. Given an object reference to an object of arbitrary type, such a universal client extracts the interface definition of the object from the IFR and dynamically constructs a user interface that reflects the operations and attributes of the object. We can then enter values into that interface; the universal client uses `DynAny` to turn these values into parameters for operations that it invokes via the DII.

17.6.2 Type Information from Translation Tables

Another option is to compose values dynamically by using rules for translating one type system into another. For example, a CORBA-CMIP bridge can use the mapping rules defined by the Joint Inter-Domain Management (JIDM) specification [24] [30] to work out how to transform each CORBA request into a Common Management Information Protocol (CMIP) request and vice versa. In

effect, you configure such a bridge by compiling the relevant IDL or GDMO[5] definitions with a tool that produces output in the form of translation tables or shared libraries to drive the operation of the bridge. The bridge uses the fixed translation rules together with the dynamic type information provided by the tool to work out how to convert requests and data types between the two protocols.

17.6.3 Type Information from Expressions

The CORBA Notification Service [26] obtains knowledge of the relevant types from its clients. Briefly, the OMG Notification Service extends the OMG Event Service (see Chapter 20) using the notion of *filters*. A filter is a Boolean expression that determines whether a particular event (which is of type `any`) will be forwarded by a channel. A client installs a filter in a channel by supplying a filter expression such as

```
$._repos_id == 'IDL:CCS/Thermostat/BtData:1.0' and
($.requested > 90 or $.requested < 20)
```

The relevant type information is supplied to the channel as part of the filter expression so that the channel can match `any` values against the filter. Typically, the channel is implemented so that it first creates an abstract syntax tree for the filter expression and then evaluates each node in the tree. Because the expression itself contains things such as repository IDs and field names, the channel can evaluate the filter against an `any` value without requiring additional type information from an interface repository.

17.7 Summary

`DynAny` provides composition and decomposition for values in a way that is analogous to the way `TypeCode` provides composition and decomposition for types. Together, `DynAny` and `TypeCode` provide the features required by generic applications that do not have knowledge of the compile-time types of values. `DynAny` was revised with CORBA 2.3 in a way that is not backward-compatible. Before developing code that uses `DynAny`, you should ensure that you have the 2.3 version.

5. GDMO stands for Guidelines for the Definition of Managed Objects. It is a type definition language for Open Systems Interconnect (OSI) network management.