Proposal for delegation based implementation which should be added after section 6.26.6 (Inheritance-Based Interface Implementation).

6.26.7 Delegation-Based Interface Implementation

Inheritance is not always the best solution for implementing servants. Using inheritance from the IDL generated traits forces a C++ inheritance hierarchy into the application. Sometimes, the overhead of such inheritance is too high, or it may be impossible to compile correctly. For example, implementing objects using existing legacy code might be impossible if inheritance from some global class were required, due to the invasive nature of the inheritance.
In some cases delegation can be used to solve this problem. Rather than inheriting from a trait, the implementation can be coded as required for the application, and a wrapper object will delegate upcalls to that implementation. This sub clause describes how this can be achieved in a type-safe manner using C++ templates. For the examples in this sub clause, the OMG IDL interface from “Inheritance-Based Interface Implementation” on page 126 will again be used.

```
// IDL
interface A
{
      short op1();
      void op2(in long val);
};
```

In addition to generating a skeleton traits, the IDL compiler generates a delegating template called a tie. This template is opaque to the application programmer, though like the skeleton, it provides a method corresponding to each OMG IDL operation. The type of the tie template is defined by the CORBA::servant_traits<T>::tie_type trait related to the corresponding interface T.

```
// C++
template<class T>
class TIE : public ….
{
public:
...
};
```

An instance of this template performs the task of delegation. When the template is instantiated with a class type that provides the operations of A, then the TIE template will delegate all operations to an instance of that implementation class. A shared pointer to the actual implementation object is passed to the tie constructor when an instance of the tie template is created. When a request is invoked on it, the tie servant will just delegate the request by calling the corresponding method in the implementation object.

```
// C++
template<class T>
class TIE : public ...
{
private:
      std::shared_ptr<T> tied_object_ {};
public:
      TIE(std::shared_ptr<T> t, IDL::traits<PortableServer::POA>::ref_type poa =
{})
: tied_object_(t), poa_(poa) {}
```

```
      virtual ~TIE() = default;
// tie-specific functions
std::shared_ptr<T> _tied_object() { return tied_object_; }
void _tied_object(std::shared_ptr<T> t)
{
tied_object_ = t;
}
// IDL operations
Short op1()
{
return tied_object_->op1();
}
void op2(Long val)
{
tied_object_->op2(val);
}
// override ServantBase operations
IDL::traits<PortableServer::POA>::ref_type _default_POA()
{
if (poa_) {
return poa_;
} else {
// return root POA
}
}
private:
IDL::traits<PortableServer::POA>::ref_type poa_;
// copy and assignment not allowed
TIE () = delete;
TIE (const TIE &) = delete;
TIE (TIE &&) = delete;
TIE& operator= (const TIE &) = delete;
TIE& operator= (TIE &&) = delete;

};
```

It is important to note that the tie example shown above contains sample implementations for all of the required functions. A conforming implementation is free to implement these operations as it sees fit, as long as they conform to the semantics in the paragraphs described below. A conforming implementation is also allowed to include additional implementation-specific functions if it wishes.

The constructors cause the tie servant to delegate all calls to the C++ object bound to shared pointer T. The _tied_object() accessor function allows callers to access the C++ object being delegated to.
For delegation-based implementations it is important to note that the servant is the tie object, not the C++ object being delegated to by the tie object. This means that the tie servant is used as the argument to those POA operations that require a Servant argument. This also means that any operations that the POA calls on the servant, such as ServantBase::_default_POA(), are provided by the tie servant, as shown by the example above. The value returned by _default_POA() is supplied to the tie constructor.
It is also important to note that by default, a delegation-based implementation (the “tied” C++ instance) has no access to the _this() function, which is available only on the tie. One way for this access to be provided is by informing the delegation object of its associated tie object. This way, the tie holds a reference to the delegation object, and vice-versa. However, this approach only works if the tie and the delegation object have a one-to-one relationship. For a delegation object tied into multiple tie objects, the object reference by which it was invoked can be obtained within the context of a request invocation

by calling PortableServer::Current::get_object_id(), passing its return value to PortableServer::POA::id_to_reference(), and then narrowing the returned object reference appropriately.

The use of templates for tie classes allows the application developer to provide specializations for some or all of the template's member functions for a given instantiation of the template. This allows the application to control how the tied object is invoked. For example, the TIE<T>::op2() operation is normally defined as follows:

```
// C++
template<class T>
void
TIE<T>::op2(Long val)
{
      tied_object_->op2(val);
}
```

This implementation assumes that the tied object supports an op2() operation with the same signature. However, if the application wants to use legacy classes for tied object types, it is unlikely they will support these capabilities. In that case, the application can provide its own specialization. For example, if the application already has a class named Foo that supports a log_value() function, the tie op2() function can be made to call it if the following specialization is provided:

```
template <>
void
CORBA::servant_traits<A>::tie_type<Foo>::op2(Long val)
{
      _tied_object()->log_value(val);
}
```

Portable specializations like the one shown above should not access tie class type and data members directly, since the names of those data members are not standardized.