Dynamic Skeleton Interface 6

The Dynamic Skeleton Interface (DSI) allows dynamic handling of object invocations. That is, rather than being accessed through a skeleton that is specific to a particular operation, an object's implementation is reached through an interface that provides access to the operation name and parameters in a manner analogous to the client side's Dynamic Invocation Interface. Purely static knowledge of those parameters may be used, or dynamic knowledge (perhaps determined through an Interface Repository) may be also used, to determine the parameters.

Contents

This chapter contains the following sections.

6.1 Introduction

The Dynamic Skeleton Interface is a way to deliver requests from an ORB to an object implementation that does not have compile-time knowledge of the type of the object it is implementing. This contrasts with the type-specific, OMG IDL-based skeletons, but serves the same architectural role.

DSI is the server side's analogue to the client side's Dynamic Invocation Interface (DII). Just as the implementation of an object cannot distinguish whether its client is using type-specific stubs or the DII, the client who invokes an object cannot determine whether the implementation is using a type-specific skeleton or the DSI to connect the implementation to the ORB.

.

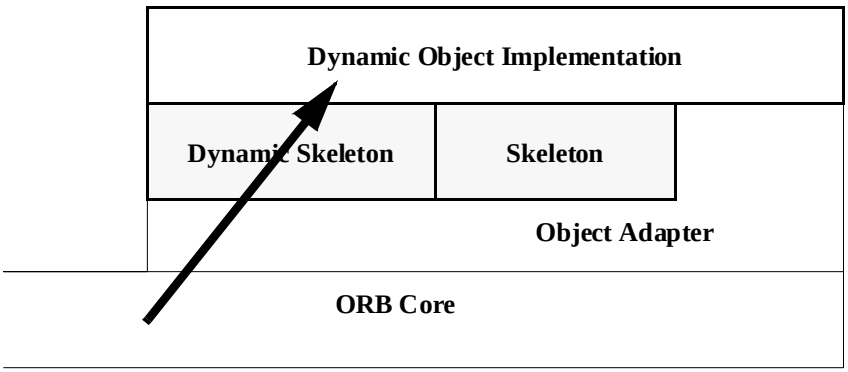

Figure 6-1 Requests are delivered through skeletons, including dynamic ones

DSI, like DII, has many applications beyond interoperability solutions. Uses include interactive software development tools based on interpreters, debuggers and monitors that want to dynamically interpose on objects, and support for dynamically-typed languages such as LISP.

6.2 Overview

The basic idea of the DSI is to implement all requests on a particular object by having the ORB invoke the same upcall routine, a Dynamic Implementation Routine (DIR). Since in any language binding all DIRs have the same signature, a single DIR could be used as the implementation for many objects, with different interfaces.

The DIR is passed all the explicit operation parameters, and an indication of the object that was invoked and the operation that was requested. The information is encoded in the request parameters. The DIR can use the invoked object, its object adapter, and the Interface Repository to learn more about the particular object and invocation. It can access and operate on individual parameters. It can make the same use of an object adapter as other object implementations.

This chapter describes the elements of the DSI that are common to all object adapters that provide a DSI. See "Single Servant, many objects and types, using DSI" on page 9-57 for the specification of the DSI for the Portable Object Adapter.

6.3 ServerRequestPseudo-Object

6.3.1 ExplicitRequest State: ServerRequestPseudo-Object

The ServerRequest pseudo-object captures the explicit state of a request for the DSI, analogous to the Request pseudo-object in the DII. The object adapter dispatches an invocation to a DSI-based object implementation by passing an instance of ServerRequest to the DIR associated with the object implementation. The following shows how it provides access to the request information:

```
module CORBA {
    ...
    pseudo interface ServerRequest {
        readonly attribute Identifier    operation;
        void          arguments(inout NVList nv);
        Context       ctx();
        void          set_result(in Any val);
        void          set_exception(in Any val);
    };
};
```

The identity and/or reference of the target object of the invocation is provided by the object adapter and its language mapping. In the context of a bridge, the target object will typically be a proxy for an object in some other ORB.

The **operation** attribute provides the identifier naming the operation being invoked; according to OMG IDL's rules, these names must be unique among all operations supported by the object's "most-derived" interface. Note that the operation names for getting and setting attributes are **_get_<attribute_name>** and **_set_<attribute_name>**, respectively. The operation attribute can be accessed by the DIR at any time.

Operation parameter types will be specified, and "in" and "inout" argument values will be retrieved, with **arguments**. Unless it calls **set_exception**, the DIR must call **arguments** exactly once, even if the operation signature contains no parameters. Once **arguments** or **set_exception** has been called, calling **arguments** on the same **ServerRequest** will result in a **BAD_INV_ORDER** system exception. The DIR must pass in to **arguments** an **NVList** initialized with **TypeCodes** and **Flags** describing the parameter types for the operation, in the order in which they appear in the IDL specification (left to right). A potentially-different **NVList** will be returned from **arguments**, with the "in" and "inout" argument values supplied. If it does not call **set_exception**, the DIR must supply the returned **NVList** with return values for any "out" arguments before returning, and may also change the return values for any "inout" arguments.

When the operation is not an attribute access, and the operation's IDL definition contains a context expression, **ctx** will return the context information specified in IDL for the operation. Otherwise it will return a nil **Context** reference. Calling **ctx** before **arguments** has been called or after **ctx**, **set_result** or **set_exception** has been called will result in a **BAD_INV_ORDER** system exception.

The **set_result** operation is used to specify any return value for the call. Unless **set_exception** is called, if the invoked operation has a non-void result type, **set_result** must be called exactly once before the DIR returns. If the operation has a void result type, **set_result** may optionally be called once with an **Any** whose type is **tk_void**. Calling **set_result** before **arguments** has been called or after **set_result** or **set_exception** has been called will result in a **BAD_INV_ORDER** system exception. Calling **set_result** without having previously called **ctx** when the operation IDL contains a context expression, or when the **NVList** passed to **arguments** did not describe all parameters passed by the client, may result in a **MARSHAL** system exception.

The DIR may call **set_exception** at any time to return an exception to the client. The **Any** passed to **set_exception** must contain either a system exception or one of the user exceptions specified in the **raises** expression of the invoked operation's IDL definition. Passing in an **Any** that does not contain an exception will result in a **BAD_PARAM** system exception. Passing in an unlisted user exception will result in either the DIR receiving a **BAD_PARAM** system exception or in the client receiving an **UNKNOWN_EXCEPTION** system exception.

See each language mapping for a description of the memory management aspects of the parameters to the **ServerRequest** operations.

6.4 DSI: Language Mapping

Because DSI is defined in terms of a pseudo-object, special attention must be paid to it in the language mapping. This section provides general information about mapping the Dynamic Skeleton Interface to programming languages.

Each language provides its own mapping for DSI.

6.4.1 ServerRequest's Handling of Operation Parameters

There is no requirement that a **ServerRequest** pseudo-object be usable as a general argument in OMG IDL operations, or listed in "orb.idl."

The client side memory management rules normally applied to pseudo-objects do not strictly apply to a ServerRequest's handling of operation parameters. Instead, the memory associated with parameters follows the memory management rules applied to data passed from skeletons into statically typed implementation routines, and vice versa.

6.4.2 Registering Dynamic Implementation Routines

In an ORB implementation, the Dynamic Skeleton Interface is supported entirely through the Object Adapter. An Object Adapter does not have to support the Dynamic Skeleton Interface but, if it does, the Object Adapter is responsible for the details.