Combat

Frank Pilhofer
fp@fpx.de

April 2, 2002

Abstract

The Combat package provides a Tcl language mapping for CORBA, allowing the implementation of CORBA clients and servers in the Tcl programming language.

On the client side, Combat is not only useful to easily test-drive your CORBA servers, but rather makes Tcl an exciting language for distributed programming. Also, with Tk, you can quickly develop attractive user interfaces accessing CORBA services. Server-side scripting using [incr Tcl] classes also offers a wide range of possibilities.

This document describes the usage of Combat and expects that you are already familiar with CORBA terminology and concepts.

The most recent version of Combat is always available from its homepage.[1]

[1] `http://www.fpx.de/Combat/`

Contents

1 Introduction

Let's begin with an example. A popular one is the "Account" example, in which a Bank supports the `create` operation to open a new account. An account, in turn, is an object that supports the `deposit`, `withdraw` and `balance` operations. The IDL file could look as follows:

```
interface Account {
  void deposit  (in unsigned long amount);
  void withdraw (in unsigned long amount);
  long balance  ();
};
interface Bank {
  Account create (in string name, in string passwd);
};
```

Assuming that a server is running, and that an object reference for the Bank is available in the file "Bank.ior" in the current directory, the following Combat script would connect the bank, open an account, deposit and withdraw some bucks, and then check its balance.

```
% set Bank [corba::string_to_object file://[pwd]/Bank.ior]
% set Account [$Bank create MyName MyPassword]
% $Account deposit 700
% $Account withdraw 450
% puts "Current balance is $Account balance]."
Current balance is 250.
```

Here's a list of Combat's main features:

- Client side Tcl scripting
- Server side Tcl scripting with full POA support
- Straightforward IDL to Tcl mapping
- Asynchronous invocations

1.1 Terminology

Before describing the individual Combat commands, we should define our terminology.

Object Reference
: An object reference, or IOR (for Interoperable Object Reference) is a *string* that uniquely identifies a server object. No operations can be performed upon an IOR directly; it must first be incarnated into a *handle* using `corba::string_to_reference`.

Handle
: A handle is a Tcl command with the same interface as the server object as defined in the IDL description. Operations (or the getting/setting of attributes) on the handle transparently cause a server invocation to happen. Handles are acquired using `corba::string_to_reference` or as a result from a method invocation.

Handles *must* be stored in a Tcl variable.[2]

Pseudo Object

A pseudo object is, on the outside, not much different from a handle. However, pseudo objects are local, and an invocation on a pseudo object does not cause a remote servant to be invoked, but is processed locally. Combat knows POA, POAManager and POACurrent pseudo objects. A pseudo object is not associated with an object reference.

Servant

Used in server-side programming. A servant is an instance of an [incr Tcl] class that inherits directly or indirectly from `PortableServer::ServantBase`. Servants are not CORBA objects; to invoke methods on a servant, one must first obtain an object reference to the servant using POA functions.

1.2 Interface Repository

The Interface Repository is vital for the operation of Combat, and it is important that you understand its importance. Combat is fully dynamic and posesses no "compile-time" knowledge of object interfaces. This is different from other language mappings, where such knowledge is generated by the IDL "compiler" in the form of stubs and skeletons.

Combat instead pulls the information from Interface Repositories, at runtime. Combat distinguishes between a *local* and many *remote* repositories. There is at most one local Interface Repository; this is the one Combat itself is connected to.[3] Then, servers themselves can be connected to different remote Interface Repositories.

For each handle, Combat must find a matching Interface Repository entry. This is done in the following way:

1. Object References *optionally* contain a Repository Id. First of all, Combat looks up that type in the *local* Interface Repository.[4]

2. If

   - the Object Reference does not contain a Repository Id,
   - there is no matching record in the local Interface Repository, or if
   - the user wants to invoke a method that is not available for the current interface and Combat wants to update its information,[5]

   a `_interface` request[6] is sent to the server. If the server is properly configured, it responds with a Interface Repository record in a *remote* Interface Repository.

[2] Even when working interactively.

[3] by performing corba::resolve_initial_references InterfaceRepository

[4] Unfortunately, there is no standard CORBA mechanism to access the Repository Id field within an object reference, so this does not work with Combat's "Generic ORB" configuration.

[5] It may be that the implementation has been replaced with a derived type.

[6] `_interface` is the GIOP name, elsewhere, it's `_get_interface`.

3. Combat uses the identifier of the remote Interface Repository's record and checks if it is also contained in the local Interface Repository, which is assumed to be more local and therefore faster. If the record is found in the local Interface Repository, Combat uses it, else it keeps the remote information.

It is a problem that, usually, administrative action is necessary for servers to properly respond to a `_interface` request. For most ORBs, this consists of the following steps:

1. Start up an Interface Repository server.[7]
2. Feed the Interface Repository with IDL files.[8]
3. Connect servers to Interface Repository.

Please see your ORB's documentation for details.

Actually, using remote Interface Repositories by questioning each object for its own interface information is the "CORBA" way to go. However, there are several reasons why you would want to use a local Interface Repository:

- You do not have administrative control over the servers, and they have not been connected to an Interface Repository, thus failing to respond to `_interface` requests.
- You have administrative control over the servers, but do not want to connect them to an Interface Repository because of overhead.
- It is faster to pull information from a local Interface Repository than from a remote one.

The local Interface Repository, if available, can be administered with the `combat::ir` command. A helper program, `idl2tcl` exists to generate a string representation of interface data which can then be used to bootstrap the local Interface Repository. See below for more information.

Some notes:

- It is assumed that data in the local Interface Repository does not conflict data in a remote Interface Repository. If an entry exists in both, its contents are assumed to be equivalent.
- For server-side scripting, all necessary interface information *must* be local.
- You have a problem if an object reference does not contain interface information *and* its server is not connected to a remote Interface Repository – if Combat does not know the object's type, it cannot look it up in the local Interface Repository. However, if you then use the `_is_a` internal operation on the handle, Combat will associate the object with that type and look it up locally.
- MICO automatically provides an empty local Interface Repository by default, if not overridden. ORBacus does not; if you do not connect to an Interface Repository upon `corba::init` and the `combat::ir` command is used, Combat autostarts an `irserv` process and from then on treats it as local.

[7] MICO: `ird`, ORBacus: `irserv`

[8] MICO: `idl --feed-ir --no-codegen-c++`, ORBacus: `irfeed`

- If you have connected to an external Interface Repository upon `corba::init`, it may be "remote" in the TCP/IP sense, but is still "local" in the Combat sense. In that case, local and remote Interface Repositories may in fact be the same.

Forgive the rather lengthy elaboration. In normal operation, you will probably never notice the fine distinction between the local and a remote Interface Repository, but the decision process is important in case anything does not work as expected.

Hints on debugging Interface Repository problems:

- Use `corba::resolve_initial_references` to check if a local Interface Repository is available. (Note that in order to perform any operations on it, the Interface Repository must contain information about the Interface Repository's interfaces.)
- The `iordump` tool that comes with many ORBs can be used to check if an object reference does include type information.
- To check if a remote server supports `_interface`, try the internal `_get_interface` operation yourself (interactively). If you get back a handle, it works, else (if you receive 0 or an exception) it doesn't.

1.3 Feature Check

At the beginning of your script, you will have to decide whether Combat supports your CORBA scripting requirements. This becomes important if you are working with multiple versions of Combat or with other CORBA scripting extensions to Tcl. The `corba::feature` command exists to verify that necessary optional features are implemented.

Syntax:

```
corba::feature names
corba::feature require ?-exact?  feature ?version?
```

The "`names`" subcommand returns a list of feature tokens supported by the implementation. For Combat, this list is *core*, *async*, *callback*, *type*, *poa*[9], *register*, *combat::ir* and *mico::bind*[10].

The "`require`" subcommand has three possible usages. If called with only a feature token, it succeeds if that feature is implemented. If that feature is not available at all, an error is returned. If a version number is mentioned, the command only succeeds if the implementation provides that feature with the same major number and at least the same minor number. With the `-exact` option, the given version number must match the implemented version exactly.

It is expected that a feature is upwards compatible within the same major version number, i.e. version 1.3 supports all operations that version 1.1 did, but version 2.1 is probably incompatible. Major version 0 is an exception in that it does not assume backwards compatibility.

As long as the Tcl language mapping is not official, Combat reports the version number of the supported features to be the same as the Combat version (i.e. less than 1.0).

[9] Only if [incr Tcl] is available

[10] Only when using the MICO ORB

1.4 Initialization

Before any of the other commands can be used, Combat and the ORB must be initialized. This is performed using the `corba::init` command.

Syntax:

```
corba::init ?arbitrary-parameters?  ?ORB-specific-parameters?
```

The command takes an arbitrary number of parameters. Combat itself does not process any parameters. They are just passed to the ORB's `CORBA::ORB_init()` method; please check your ORB's manual for a listing of potential options. The ORB will consume all ORB-specific arguments and remove them from the command line, the remaining parameters are returned.

It's a good idea to pass a script's command-line arguments, which are contained in the `argv` variable, to `corba::init`. This can be done using

```
set argv [eval corba::init $argv]
```

Afterwards, `argv` contains the remaining options.

Calling `corba::init` is optional. If not performed explicitly, it is invoked implicitly with an empty list of arguments if the ORB is first accessed through any of the other commands.[11]

2 Client Side scripting

2.1 idl2tcl

As already mentioned above, Combat uses not precompiled stubs but the Dynamic Invocation Interface for method invocations. In order to construct a request, information about the available methods and the types of their parameters is needed. Combat reads this information from an Interface Repository to typecheck parameter values.

A standalone program, `idl2tcl` is provided that reads in IDL files and that produces Tcl declarations that can then be fed into the local Interface Repository with the `combat::ir` command (see below).

Usage:

```
idl2tcl ?--name name?  idl-file ...
```

The resulting Tcl script is written to a file with the same base name as the IDL file, but with the `.tcl` extension, in the current directory. It can then be read in your own scripts with the `source` command, or appropriate portions can be copied and pasted.

The generated script initializes the Tcl variable `_ir_`*name* to contain appropriate definitions for the Interface Repository that can be used in a call to `combat::ir add`. In the above variable name, *name* is the base name of the last IDL input file on the command line, or the parameter given to the `--name` option.

Suppose you had a simple IDL file `hello.idl`,

[11] Except `corba::feature`, which does not require the ORB.

```
interface HelloWorld {
  void hello ();
};
```

You could then "compile" the file to a Tcl definition using

```
idl2tcl hello.idl
```

In your own scripts, you would initialize the local Interface Repository using

```
source hello.tcl
combat::ir add $_ir_hello
```

before connecting to any `HelloWorld` objects.

You may find it interesting that `idl2tcl` is itself a Combat program that browses the Interface Repository using self-generated type information.

2.2 ORB Methods

Some basic ORB methods are provided in the `corba` namespace. For details, see the CORBA specification, Chapter 4 ("ORB Interface").

`corba::string_to_object` *`ior`*

Takes an object reference string as parameter and incarnates into a new handle.

`corba::object_to_string` *`handle`*

Takes a handle as parameter and returns the stringified IOR of the associated object reference.

`corba::resolve_initial_references` *`id`*

Obtains an initial reference and incarnates it into a new handle. Examples include "RootPOA", "POACurrent", "InterfaceRepository" and "NameService". For "RootPOA" and "POACurrent", a pseudo object rather than a "normal" handle is returned.

Note that to access the Interface Repository or the Naming Service, the Interface Repository must contain appropriate information about the associated interface.

`corba::list_initial_services`

Returns a (potentially incomplete) list of valid ids that can be used with `corba::resolve_initial_references`.

2.3 The MICO Binder

Only supported when Combat is running with MICO.

To access a service, you need some mechanism to receive its object references. This can be done by passing IOR strings, or via the Naming Service. The MICO Binder [12] can be thought of as a very simple naming mechanism, it searches for a server based on its Repository Id.

```
mico::bind ?-addr addr?  repoid ?Tag?
```

If no explicit address is given with the `-addr` option, all remote ORBs that were given upon initialization (with the `-ORBBindAddr` option) are contacted and asked for a server serving the given Repository Id.

If no `Tag` (an arbitrary string that may be used to distringuished different servers serving the same interface) is given, any such server is acceptable; otherwise, a server object with the same tag is searched for.

Because this mechanism is MICO-specific, the command resides in the `mico::` namespace. You can only bind to servers implemented with MICO.[13]

2.4 Handles

As already noted, a handle is a Tcl command which you can invoke available operations, set or query attributes on, as specified in the IDL description for that particular interface.

Method invocations:	`$obj` *op* `?parameters ...?`
Query attribute:	`$obj` *attribute*
Set attribute:	`$obj` *attribute value*

Invocations are usually synchronous and will wait until the result from the server is available. See *Asynchronous Invocations* for information about asynchronous invocations.

An operation is mapped to a Tcl procedure with the same number of parameters as in the IDL interface description. `in` parameters are passed *by value*, as expected, while `out` and `inout` parameters are passed *by reference*. Consider the operation

```
interface A {
  short op (in long val, inout short flags, out string name);
};
```

To invoke `A::op`, you pass the first parameter by value, while you must put the second parameter in a variable first, and must give a variable name for the third parameter:

```
set flags 42
set res [$Aobj op -1 flags name]
```

Note that we did not use `$flags` or `$name`, and didn't need to set the `name` variable prior to the invocation. Afterwards, you will find in `flags` and `name` the values returned by the operation.

[12] See also the MICO manual.

[13] This includes Combat servers, if Combat is built upon MICO.

Please see “The IDL to Tcl mapping” below for the details about how CORBA data types are mapped to Tcl.

Each handle also supports the following “builtin” operations which have the same semantics as defined in the CORBA specification.

`_get_interface`
: Returns a handle of type `CORBA::InterfaceDef` pointing into the Interface Repository. Since the interface type for handles must be known, this requires that the IDL description for the Interface Repository has been loaded into the Interface Repository itself.

`_is_a` ***repoid***
: Takes a Repository Id as parameter and returns true (1) if the object implements the given interface.

`_non_existent`
: Returns true (1) if the server providing the implementation for this object has ceased to exist. A false return value (0) does not guarantee that any following invocations will succeed.

`_is_equivalent` ***handle***
: Takes another handle as parameter and returns true (1) if the objects referenced by both handles are equivalent, or false (0) if not.

`_duplicate`
: Returns a duplicate of the handle. See the section about handle management for more information.

Note that there is no need for an “is_nil” operation, because nil object references are never incarnated into a handle.

2.5 Asynchronous Invocations

As described so far, method (or attribute) invocations are synchronous, and the invocation blocks until the result (or a success message) is received from the server. Additional flags can be added *before the attribute or operation name* to make an invocation asynchronous.

```
$obj -async op ?parameters ...?
$obj -callback proc op ?parameters ...?
```

With `-async` or `-callback`, the invocation does not wait for the result, but returns immediately. Instead of the operation’s result, an *async-handle* is returned.

`-callback` arranges for the given procedure to be called once the server process returns and the result becomes available. The procedure is called at global level with the handle as single parameter.

The `corba::request` command exists to monitor the status of asynchronous invocations in progress.

Syntax:

```
corba::request get handle
corba::request poll ?handle ...?
corba::request wait ?handle ...?
```

`get`
: Waits until the asynchronous request with the given async-handle has finished, and returns the result of the operation, or throws an exception in case of a failure. Also extracts any `out` or `inout` parameters *within the context of the* `get` *invocation* (see below).

`poll`
: If called without arguments, it checks if any of the currently active asynchronous invocations has finished. If `poll` is called with one or more handles as arguments, it checks if any of these has finished. If yes, a single handle is returned. `get` should then be called on that handle to retrieve the result. If none of the (given) request has finished, `poll` returns immediately with an empty result.

`wait`
: Similar to `poll`, but waits until one request has finished and then returns its handle. If called without arguments, it considers all asynchronous requests that are in progress. If there are no outstanding asynchronous requests, it immediately returns with an empty result.

A callback procedure receives a handle as single argument and is expected to perform a `corba::request get` on that handle. Here's a simple example for a callback:

```
proc cb {handle} {
  set res [corba::request get $handle]
  puts "Result is: $res"
}
$obj -callback cb sleep 10
```

You must be careful using asynchronous invocations for operations with `out` or `inout` parameters. When setting up the invocation, only the name of the variable that was given for the `out` or `inout` parameter is stored, and they are written to in the context in which the corresponding `corba::request get` is executed. So unless you declare the variables global inside a callback function, they will not be visible on the outside. Here's an example. Imagine an object with a `strcpy` procedure that takes as parameters an `out` string (dest), and an `in` string.

```
proc cb {handle} {
  global dest
  corba::request get $handle
}
global dest
$obj -callback cb strcpy dest "Hello World"
vwait dest
```

If `corba::request get` is executed in the callback procedure, the `dest` variable, which is declared to take the `out` string parameter to `strcpy`, is set. If `dest` were not declared global, it would be set locally, and the `vwait` would block forever.

Notes:

- The `-async` and `-callback` flags can be used likewise on operations and attributes (for very remote servers, setting or retrieving an attribute may take some time).
- Ordering is not guaranteed for asynchronous invocations, not even on the same object.
- Asynchronous invocations are only processed in Tcl's event loop, so if your application isn't event driven, make sure to call `update` or `vwait` once in a while.
- Pseudo objects support the same syntax for asynchronous invocations. However, the asynchrony is just "simulated" – operations on pseudo objects always happen synchronously when setting up the request.

2.6 Accessing Const Values

Constant values (declared with the IDL keyword `const`) can be accessed with the `corba::const` command:

```
corba::const repoid-or-scoped-name
```

Looks up the constant in the *local* Interface Repository using either its Repository Id or its scoped name and returns the constant's value as an `Any` value.

2.7 Handle Management

There are two commands related to duplicating and releasing handles. There are subtle differences in handle management depending on which version of Combat you are using. In Combat/C++, handle management is fully *automatic*, and you need not spend much thought on it. In Combat/Tcl, handle management is *manual*.

Combat/C++ automatically releases all memory that is associated with a handle if the handle is no longer referenced by a Tcl variable. In Combat/Tcl, you must use `corba::release` in order to release all memory. If you want to keep a handle even though it will be released elsewhere, you must use `corba::duplicate` to create a copy. One popular example where duplicates are needed is in a servant, which receives an object reference as parameter. Because the handle that is passed as a parameter will be released by the runtime after the servant's method returns, the servant must create a duplicate in order to keep a copy.

```
corba::duplicate ?typecode?  value
```

The `corba::duplicate` command takes a value as parameter, and optionally a typecode. If the typecode is omitted, then *value* must be a handle. This handle is then duplicated, and a new handle that encapsulates the same object reference as the original, is returned. If a typecode is present, then *value* must match that typecode. The command will then traverse the value according to the typecode and duplicate all of its handles. A "deep copy" is then returned.

```
corba::release ?typecode?  value
```

The syntax of `corba::release` is the same as `corba::duplicate`. If the typecode is omitted, then *value* must be a handle. All memory that the ORB associates with this handle is released, and no further invocations using this handle are possible. If a typecode is present, then *value* must match that typecode. The command will then traverse the value according to the typecode and release all of its handles.

Since handle management in Combat/C++ is automatic, both of these two commands exist in Combat/C++ for compatibility only.

2.8 Dynamic Invocations

Invocations normally pull type information from the Interface Repository. In contrast, an invocation using **corba::dii** does not require type information for the remote interface to be present in the Interface Repository; here, type information is passed along with each invocation in a separate *spec* parameter:

```
corba::dii handle spec ?parameters ...?
```

spec is a list composed of three or four elements. The first element is the typecode of the return value. The second element is the name of the operation to be invoked. The third element describes the parameters. The fourth element is a list of exception typecodes that this operation may throw. The parameter description is a list that contains one element per parameter. Each parameter is described by a list of two elements. The first element is either **in**, **out** or **inout**, and the second element is the typecode of the parameter type.

As described in the section about asynchronous invocations, you can also use the **-async** or **-callback** option to initiate a dynamic invocation asynchronously.

3 The IDL to Tcl mapping

3.1 Mapping of Data Types

This section describes how IDL data types are mapped to Tcl types.

Primitive Types

`short`, `long`, `unsigned short`, `unsigned long`, `long long` and `unsigned long long` values are mapped to Tcl's integer type. Errors may occur if a value exceeds the numerical range of Tcl's integer type.

`float`, `double`, `long double` values are mapped to Tcl's floating point type.

`string` and `wstring` values are mapped to Tcl strings.

`boolean` values are accepted as 0, 1, true, false, yes and no. In a result, they are always rendered as 0 (false) and 1 (true).

`octet`, `char` and `wchar` values are mapped to strings of length 1.

`fixed` values are mapped to a floating-point value in exponential representation. Depending on their scale and value, it may or may not be possbile to use the value in a Tcl expression.

Struct Types

`struct` values are mapped to a list. For each element in the structure, there are two elements in the list – the first is the element name, the second is the element's value. This allows to easily assign structures from and to associative arrays, using `array get` and `array set`.

Example: the IDL type

```
struct A {
  unsigned long B;
  string C;
};
```

can be matched by the Tcl list {`B 42 C {Hello World}`}.

Sequences

`sequence` values are mapped to a list. As an exception, sequences of `char`, `octet` and `wchar` are mapped to strings.

Example: the IDL type (following the above example for a structure)

```
typedef sequence<A, 2> D;
```

can be matched by the Tcl list {{`B 42 C {Hello World}`}}. Note the extra level of nesting compared to the struct above.

Arrays

`array` values are mapped to a list. As an exception, sequences of `char`, `octet` and `wchar` are mapped to strings.

Enumerations

`enum` values are mapped to the enumeration identifiers (without any namespace qualifiers).

Example: the IDL type

```
enum E {F, G, H};
```

can be matched by the Tcl string `G`.

Unions

`union` values are mapped to a list of length 2. The first element is the discriminator, or `(default)` for the default member. The second element is the appropriate union member. Note that the default case can also be represented by a concrete value distinct from all other case labels.

Object References

Non-nil object references are mapped to handles. Nil object references are mapped to the integer value 0 (zero).

Exceptions

`exception` values are mapped to a list of length one or two. The first element is the Repository Id for the exception. If present, the second element is the exception's contents, equivalent to the structure mapping. The second element may be omitted if the exception has no members.

Value Types

`valuetype` values are mapped to a list, like `structs`. For each element in the inheritance hierarchy of a `valuetype`, there are two elements in the list – the first is the element name, and the second is the element's value. An additional member `_tc_` may be present. If present, its value must be a typecode. In an invocation, this member determines the type to be sent. This mechanism allows to send a derived valuetype where a base valuetype is expected. If no `_tc_` member is present, the valuetype must be of the same type as requested by the parameter. In receiving a valuetype, the _tc_ member is always added. A `valuetype` can also be the integer 0 (zero) for a null value.

Note that this language mapping disallows valuetypes that contain themselves.

`custom` valuetypes are not supported.

Value Boxes

Boxed `valuetype` types are mapped to either the boxed type or to the integer 0 (zero) for a null value. In the case of boxed integers, the value 0 will always be read as a null value rather than a non-null value containing the boxed integer zero. Shoot yourself in the foot if you run into this problem.

TypeCode values

`TypeCode` values are mapped to a string containing a description of the typecode:

- Typecodes for the primitive types `void`, `boolean`, `short`, `long`, `unsigned short`,`unsigned long`, `long long`, `unsigned long long`, `float`, `double`, `long double`, `char`, `octet`, `string`, `any`, `TypeCode` are mapped to their name.
- Bounded string typecodes are mapped to a list of length two. The first element of the list is the identifier `string`, the second element is the bound.
- Bounded wstring typecodes are mapped to a list of length two. The first element of the list is the identifier `wstring`, the second element is the bound.
- `struct` typecodes are mapped to a list of length three. The first element is the identifier `struct`. The second element is the Repository Id, if available (else, the field may be empty). The third element is a list with an even number of elements. The zeroth and other even-numbered elements are member names, followed by the member's typecode.

- `union` typecodes are mapped to a list of length four. The first element is the identifier `union`. The second element is the Repository Id, if available (else, the field may be empty). The third element is the typecode of the discriminator. The fourth element is a list with an even number of elements. The zeroth and other even-numbered elements are labels or the identifier `default` for the default label, followed by the typecode of the associated member.
- `exception` typecodes are mapped to a list of length three. The first element is the identifier `exception`, the second element the Repository Id, and the third element is a list with an even number of elements. The zeroth and other even-numbered elements are member names, followed by the member's typecode.
- `sequence` typecodes are mapped to a list of length two or three. The first element is the identifier `sequence`, the second element is the typecode of the member type. The third element, if present, denotes the sequence's bound. Otherwise, the sequence is unbounded.
- `array` typecodes are mapped to a list of length three. The first element is the identifier `array`, the second element is the typecode of the member type, the third element is the array's length.
- `enum` typecodes are mapped to a list of length two. The first element is the identifier `enum`, the second element is a list of the enumeration identifiers.
- Object reference typecodes are mapped to a list of length two. The first element is the identifier `Object`, the second element is the Repository Id of the IDL `interface`.
- `fixed` typecodes are mapped to a list of length three. The first element is the identifier `fixed`. The second element is the number of significant digits, the third element is the scale.
- `valuetype` typecodes are mapped to a list of length five. The first element is the identifier `valuetype`. The second element is the Repository Id. The third element is a list of non-inherited members. For each member, there are three elements in the list, a visibility (`private` or `public`), the member name and the member's typecode. The fourth element is the typecode of the valuetype's concrete base, or 0 (zero) if the valuetype does not have a concrete base. The fifth element is either an empty string or one of the modifiers `custom`, `abstract` or `truncatable`.
- Boxed `valuetype` typecodes are mapped to a list of length 3. The first element is the identifier `valuebox`. The second element is the Repository Id, and the third element is the typecode of the boxed type.
- A recursive reference to an outer type (in a `struct`, `union` or `valuetype`) can be expressed by a list of length two. The first element is the identifier `recursive`, the second element is the Repository Id of the outer type, which must appear in the same typecode description.

Examples for legal TypeCodes are:

- `struct {} {s short ul {unsigned long} Q string}`

- `enum {A B C}`
- `union {} short {0 boolean (default) string}`
- `struct IDL:S:1.0 {foo {sequence {recursive IDL:S:1.0}}}`

See the description of `corba::type`. The `of` subcommand can be used to retrieve TypeCode information from the Interface Repository, the `equivalent` subcommand can be used to check TypeCode values against known types.

Any values

`any` values are mapped to a list of length two. The first element is the typecode, and the second element is the value.

3.2 Exceptions

3.2.1 Throwing Exceptions

Exceptions can be thrown with the `corba::throw` command.

Syntax:

```
corba::throw <exception>
```

The parameter must be an exception according to the mapping above: a list of length one or two. The first element of the list is the exception's Repository Id. The second element is a list of the exception's members according to the mapping for structures. If the following two exceptions were declared in interface A,

```
interface A {
  exception EX {
    long value;
    string reason;
  };
  exception OOPS {};
};
```

then these would be legal invocations of `corba::throw`

```
corba::throw {IDL:A/EX:1.0 {value 42 reason "oops, what's up?"}}
corba::throw IDL:A/OOPS:1.0
```

The exception must be known to the *local* Interface Repository, where the given Repository Id is looked up.[14] In the second example, the exception's members could be omitted because the exception does not have any members.

If this command is used in a servant in the context of a server invocation, and if the exception is not caught within the servant, it is passed back to the client side. If an exception is not caught within a client, the client prints an error message and terminates.

[14] With the exception of system exceptions.

3.2.2 Catching Exceptions

Exceptions can be caught with Tcl's `catch` command. See the Tcl manual page for details.

```
catch {
    ... object invocations that might throw exceptions ...
} result
```

If this command returns 0, the script has completed successful, without throwing an exception, and result contains the script's return code. If `catch` returns 1, an error has happened, and the error code is stored in the result variable.

A small problem is that not all errors are exceptions, so you will have to check the result from `catch` carefully if it contains an exception or other error information. You should check the first item of the result if it is a known exception's Repository Id.

For convenience, you can also use the `corba::try` command to handle exceptions. It implements Java-style processing of exceptions.

Syntax:

```
corba::try block ?catch {repoid ?var?} c-block? ... ?finally f-block?
```

First, the code block `block` is evaluated. If a CORBA exception or Tcl error has occured, then the `catch` clauses are searched left to right. Each `catch` clause is associated with a Repository Id and a code block. For the first clause whose Repository Id matches the Repository Id of the exception that has occured, the associated code block is executed.

The special value `...` for a Repository Id in a `catch` clause is recognized to match all CORBA exceptions and Tcl errors.

If a variable name is associated with the Repository Id in a `catch` clause, this variable is set to the exception that has occured during execution of the associated code block.

Regardless of whether an exception or error has occured and whether an exception or error has indeed been handled by a `catch` clause, the code block associated with the `finally` clause is, if it exists, always executed.

If there are no `catch` clauses, an implicit clause that catches `...` is used.

The return value of the `corba::try` statement is computed as follows, in order of priority:

- If a `finally` clause exists and its associated code block completes with a return value different from `TCL_OK` (i.e. causes itself an error or executes a Tcl `return`, `break` or `continue` statement), then this return code is used.
- If a CORBA exception or Tcl error occurs while executing the "main" code block, and this exception or error is handled by a `catch` clause, then the return value of the associated code block for this clause is used.
- If a CORBA exception or Tcl error occurs while executing the "main" code block, and this exceptions is not handled by a `catch` clause, then this error is used.
- If no CORBA exception or Tcl error occurs while executing the "main" code block, then its return code is used.

One effect of this return value handling is that all code blocks may execute a Tcl `return`, `break` or `continue` statement, which will then be correctly passed along to the surrounding code.

3.2.3 Exception Example

Here's an example how exceptions could be handled:

```
corba::try {
   ...
} catch {IDL:A/EX:1.0 oops} {
  # oops contains A::EX data
} catch {IDL:omg.org/CORBA/COMM_FAILURE:1.0} {
  # remote server may be down
} catch {... oops} {
  puts "oops: unexpected exception: $oops"
}
```

3.3 Working with TypeCodes

The `corba::type` command can be used to ensure type safety.

`corba::type of <repoid-or-scoped-name>`
: When given the Repository Id or scoped name of any type (such as a struct), this command retrieves the type code from the *local* Interface Repository.

`corba::type match <TypeCode> <value>`
: If the value matches the type code, this command returns 1, otherwise 0. This enables applications to verify type-safety of operation parameters or type-safe composition of Any values.

`corba::type equivalent <TypeCode> <TypeCode>`
: Type codes can be compared for identity using string comparison. This method is a less strict comparison and returns 1 if both types are equivalent, meaning that they accept the same values (for example, they ignore type aliasing). This information can for example be used to extract values from an Any, by comparing the Any's type code against known (expected) type codes.

4 The Interface Repository

Combat provides the `combat::ir` command to access the Interface Repository, which resides in the `combat` namespace because it is specific to the Combat package. You have already learned one usage of this command when bootstrapping a local Interface Repository with information about the interfaces generated by `idl2tcl`.

Syntax:

```
combat::ir add ir-description-seq
```

This adds new entries to the *local* Interface Repository. `ir-description-seq` must be a string generated by `idl2tcl`. In case of duplicates, existing entries in the Interface Repository are overwritten, while modules and interfaces are reopened and added to.

5 Server Side Scripting

5.1 Implementing Servants

The server-side mapping is kept similar to the C++ mapping in that it is based on the POA and associated interfaces. Servants are realized using [incr Tcl] classes. Previous exposure to [incr Tcl] is useful, but not required.

To implement a servant, you must write an [incr Tcl] class that inherits, directly or indirectly, from the Combat-provided class `PortableServer::ServantBase`. In the implementation, you must provide public variables for IDL attributes and a public method for IDL operations, all with the same name as in the IDL file.

As a little piece of magic, since Combat does not have compile-time type information, you must provide run-time type information. This is done by implementing the public method `_Interface` (leading underscore, capital I), which does not have any parameters and must return the Repository Id for the servant's most-derived IDL interface.

> Important note: Type information for all interfaces *must* be contained in the *local* Interface Repository!

As an example, consider the following IDL file:

```
interface Foo {
  void HelloWorld ();
  attribute short x;
};
```

Its implementation could look like

```
class Foo {
  inherit PortableServer::ServantBase

  public method _Interface {} {
    return "IDL:Foo:1.0"
  }

  public method HelloWorld {} {
    puts "Hello World"
  }

  public variable x
}
```

As with method invocations, `in` parameters are passed by value, while `out` and `inout` parameters are passed by reference. Consider the operation

```
interface A {
  short op (in long val, inout short flags, out string name);
};
```

In your implementation, you receive variable names for the `flags` and `name` parameters. However, since these variables are set "outside" your class method, i.e. one level "above", you must "import" them using `upvar` (see the Tcl manual). Therefore, an implementation for the above method could look like

```
class A {
  inherit PortableServer::ServantBase

  public method _Interface {
    return "IDL:A:1.0"
  }

  public method op { val flags_name name_name } {
    upvar $flags_name flags $name_name name
    puts "val is $val"
    puts "flags is $flags"
    set flags -1
    set name "Hello World"
    return 42
  }
}
```

Now that we have written an implementation, we can create an instance of that class ("Servant") using

```
set serv [Foo #auto]
```

Servant memory mangement is left entirely to the user. Servants are allocated and deleted using the above construction mechanism and [incr Tcl]'s `delete` operator. The application is responsible not to delete any servants that are still referenced in a POA.

Servants are not automatically accessible from the outside after their creation. They must be `activated` in a POA first.

Each servant inherits the `_this` member function, which has three purposes (the same as in the C++ mapping).

1. Within the context of a request invocation, returns a new handle incarnating a reference for the object that the servant currently incarnates.

2. Outside the context of a request invocation, if the servant has not yet been activated, and if its POA has the `IMPLICIT_ACTIVATION` policy, the servant is activated, and a handle incarnating an object reference to that servant is returned.

3. Outside the context of a request invocation, if the servant has already been activated, and if its POA has the `UNIQUE_ID` policy, a handle incarnating an object reference to that servant is returned.

5.2 The POA Pseudo Object

A pseudo object for the Root POA is obtained using `corba::resolve_initial_references`:

```
set RootPOA [corba::resolve_initial_references RootPOA]
```

POA pseudo objects support all operations as defined in the CORBA specification. The usual type mapping rules apply, with a single exception. The `create_POA` method receives as its second parameter a list of policy *values* rather than a list of policy *objects*. That means that the corresponding factory operations, like `create_lifespan_policy` are not needed. Example:

```
set myPOA [$RootPOA create_POA 0 {USE_SERVANT_MANAGER PERSISTENT}]
```

This creates a new POA as a child of the Root POA. A new POA Manager is created, because a nil value rather than a handle is passed as the first parameter. The new POA will support persistent objects and use a servant manager.

The "native" data types from the POA specification are represented in the following way:

PortableServer::Servant
: Servants are instances of an [incr Tcl] class that derives from `PortableServant::ServantBase`, as seen above.

PortableServer::ObjectId
: ObjectIds are mapped to Tcl strings.

PortableServer::ServantLocator::Cookie
: Cookies are mapped to Tcl strings.

5.3 The POA Current Pseudo Object

A POA Current pseudo object is obtained using `corba::resolve_initial_references`.

```
set POACurrent [corba::resolve_initial_references POACurrent]
```

A POA Current pseudo object implements all operations as defined in the CORBA specification:

`get_POA`
: In the context of a method invocation on a servant, returns the POA in whose context it is called.

`get_object_id`
: In the context of a method invocation, returns the Object Id identifying the object in whose context it is called.

5.4 The POA Manager Pseudo Object

A POA Manager pseudo object is obtained using the `the_POAManager` method on a POA pseudo object. It implements the following methods as defined in the CORBA specification:

`activate`
Switches all associated POAs to the "active" state so that they can start serving requests.

`hold_requests` ***wait_for_completion***
Switches all associated POAs to the "holding" state, so that incoming method invocations are queued. Queued requests are performed when the POA again enters the active state.

`discard_requests` ***wait_for_completion***
Switches all associated POAs to the "discarding" state, so that incoming method invocations are discarded rather than processed.

`deactivate` ***etherealize wait_for_completion***
Switches all associated POAs to the "inactive" state. If *etherealize* is true, a servant manager, if available, is asked to "etherealize" active objects.

5.5 Examples

While implementing servants should be pretty straightforward, the number of possibilities for handling servants with the POA is pretty confusing. Let's write a simple servant, and then try a few examples. In the following examples, we assume that you have "compiled" the IDL file into a Tcl file using `idl2tcl`, that you have sourced that file and fed the local Interface Repository.

The "Hello World" IDL definition:

```
interface HelloWorld {
  void hello ();
};
```

The "Hello World" implementation:

```
class HelloWorld {
  inherit PortableServer::ServantBase

  public method _Interface {} {
    return "IDL:HelloWorld:1.0"
  }

  public method hello {} {
    puts "Hello World"
  }
}
```

Now, the following few lines of code create a Hello servant, activate it with the POA and starts serving request.

```
set poa [corba::resolve_initial_references RootPOA]
set mgr [$poa the_POAManager]

set serv [Hello #auto]

$poa activate_object $serv
$mgr activate
vwait forever
```

First, we obtain the POA and POAManger pseudo objects. Then, we create an instance of the "Hello" class and activate it using the `activate_object` method on the POA. Then, we use `activate` on the POA Manager to transition the POA from its initial Holding to the active state. Last, we enter Tcl's event loop by waiting for the `forever` variable to change, which never happens – so Tcl never returns from the event loop and will process requests forever.

As an alternative to `activate_object`, we could also use `servant_to_id`. Since the Root POA has the `IMPLICIT_ACTIVATION` policy, it would cause the servant to be implicitely activated.

Another alternative is to call the servant's inherited `_this` member function, which also implicitely activates a servant.

Now let's assume we want to activate more than one Hello servant, and we want to assign each servant an Object Id of our choice, so that clients can bind to a specific servant. Since the RootPOA has the `SYSTEM_ID` policy, this involves creating a new POA that has the `USER_ID` policy.

```
set poa   [corba::resolve_initial_references RootPOA]
set mgr   [$poa the_POAManager]
set mypoa [$poa create_POA MyPOA $mgr {USER_ID}]

set serv1 [Hello_impl #auto]
set serv2 [Hello_impl #auto]

$mypoa activate_object_with_id Hello-1 $serv1
$mypoa activate_object_with_id Hello-2 $serv2
$mgr activate
vwait forever
```

For a more complex example, see the server of the Bank/Account example in the `demo/account` subdirectory. There, `create_reference_with_id` is used in the Bank factory to create references to non-existent Account objects. A Servant Activator is then registered to create Accounts on demand.

5.6 Limitations

Because [incr Tcl] currently does not support virtual inheritance, Combat does not support multiple inheritance yet.

However, single implementation inheritance works, you can simply inherit from the base implementation instead of `PortableServer::ServantBase`.

6 To Do

Combat seems reasonably complete. Some random leftover thoughts:

- Multithreading is not yet supported. It might work if Combat commands are only used from a single thread, but this is untested. If multithreading was supported, would it eliminate the need for asynchrony?
- Should [incr Tcl] be replaced on the server side? It's basically nice, but does not support diamont inheritance, and does not allow for reference-counted objects.
- Maybe interface information could be stored elsewhere than in the Interface Repository, for example by storing `FullInterfaceDescription` data. This would improve things with ORBs that do not provide an "internal" IFR.
- It would be wonderful to submit the interfaces and mappings in this document as an official OMG language mapping. I do not have the authority and muscle to do that on my own.

Note that the author's motivation in further development of Combat is partly fueled by user feedback. I would love to hear of projects using Combat, or of plans to use it.