JacORB 2.3.0.4

IDL Compiler and Interface Repository

Bundled for OpenFusion RTOrb Java Edition

The JacORB Team

January 14, 2011

Contributors in alphabetical order:

Alphonse Bendt
Gerald Brose
Nick Cross
Phil Mesnier
Nicolas Noffke
Steve Osselton
Simon McQueen
Francisco Reverbel
David Robison
André Spiegel

Contents

1 Interface Repository

Run–time type information in CORBA is managed by the ORB's *Interface Repository* (IR) component. It allows to request, inspect and modify IDL type information dynamically, e.g., to find out which operations an object supports. Some ORBs may also need the IR to find out whether a given object's type is a subtype of another, but most ORBs can do without the IR by encoding this kind of type information in the helper classes generated by the IDL compiler.

In essence, the IR is just another remotely accessible CORBA object that offers operations to retrieve (and in theory also modify) type information.

1.1 Type Information in the IR

The IR manages type information in a hierarchical containment structure that corresponds to the structure of scoping constructs in IDL specifications: modules contain definitions of interfaces, structures, constants etc. Interfaces in turn contain definitions of exceptions, operations, attributes and constants. Figure 1.1 illustrates this hierarchy.

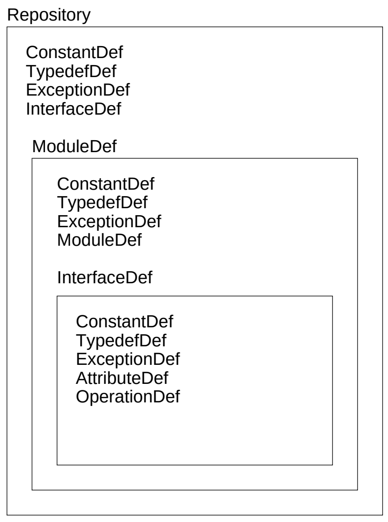


Figure 1.1: Containers in the Interface Repository

The descriptions inside the IR can be identified in different ways. Every element of the repository has a unique, qualified name which corresponds to the structure of name scopes in the IDL specification. An interface `I1` which was declared inside module `M2` which in turn was declared inside module `M1` thus has a qualified name `M1::M2::I1`. The IR also provides another, much more flexible way of naming IDL constructs using *Repository Ids*. There are a number of different formats for RepositoryIds but every Repository must be able to handle the following format, which is marked by the prefix `"IDL:"` and also carries a suffix with a version number, as in, e.g., "`IDL:jacorb/demo/grid:1.0`". The name component between the colons can be set freely using the IDL compiler directives `#pragma prefix` and `#pragma ID`. If no such directive is used, it corresponds to the qualified name as above.

1.2 Repository Design

When designing the Interface Repository, our goal was to exploit the Java reflection API's functionality to avoid having to implement an additional data base for IDL type descriptions. An alternative design is to use the IR as a back-end to the IDL compiler, but we did not want to introduce such a dependency and preferred to a have a rather "light–weight" repository server. As it turned out, this design was possible because the similarities between the Java and CORBA object models allow us to derive the required IDL information at run time. As a consequence, we can even do without any IDL at compile time. In addition to this simplification, the main advantage of our approach lies in avoiding redundant data and possible inconsistencies between persistent IDL descriptions and their Java representations, because Java classes have to be generated and stored anyway.

Thus, the Repository has to load Java classes, interpret them using reflection and translate them into the appropriate IDL meta information. To this end, the repository realizes a reverse mapping from Java to IDL. Figure 1.2 illustrates this functionality, where f^{-1} denotes the reverse mapping, or the inverse of the language mapping.

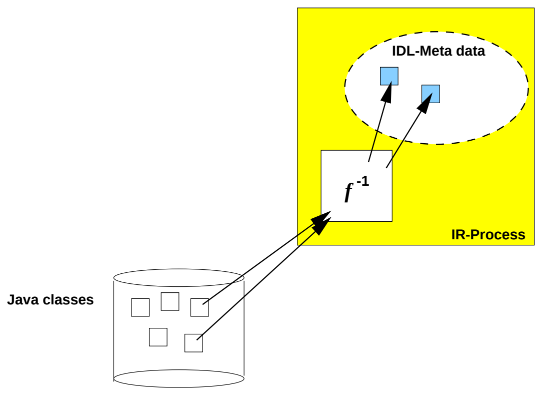


Figure 1.2: The JacORB Interface Repository

1.3 Using the IR

For the ORB to be able to contact the IR, the IR server process must be running. To start it, simply type the `ir` command and provide the required arguments:

```
$ ir /home/brose/classes /home/brose/public_html/IR_Ref
```

The first argument is a path to a directory containing `.class` files and packages. The IR loads these classes and tries to interpret them as IDL compiler–generated classes. If it succeeds, it creates internal representations of the adequate IDL constructs. See below for instructions on generating classes with IR information. The second argument on the command line above is simply the name of the file where the IR stores its object reference for ORB bootstrapping.

To view the contents of the repository, you can use the GUI IRBrowser tool or the query command. First, let's query the IR for a particular repository ID. JacORB provides the command `qir` ("query IR") for this purpose:

```
$ qir IDL:raccoon/test/cyberchair/Paper:1.0
```

As result, the IR returns an InterfaceDef object, and `qir` parses this and prints out:

```
    interface Paper
    {
       void read(out string arg_0);
       raccoon::test::cyberchair::Review getReview(in long arg_0);
       raccoon::test::cyberchair::Review submitReview(
           in string arg_0, in long a rg_1);
       void listReviews(out string arg_0);
    };
```

To start the IRBrowser, simply type

```
$ irbrowser [ -i <IOR-string> | -f <filename>]
```

e.g.

```
$ irbrowser
```

Note that if no arguments are supplied it will default to using resolve_initial_references.

Figure 1.3 gives a screen shot of the IR browser.

The Java classes generated by the IDL compiler using the standard OMG IDL/Java language mapping do not contain enough information to rebuild all of the information contained in the original IDL file. For example, determining whether an attribute in an interface was `readonly` or not is not possible, or telling the difference between `in` and `inout` parameter passing modes. Moreover, IDL modules are not explicitly represented in Java, so telling whether a directory in the class path represents an IDL module is not easily possible. For these reasons, the JacORB

IDL compiler generates a few additional classes that hold the required extra information if the compiler switch `-ir` is used when compiling IDL files:

```
$ idl -ir myIdlFile.idl
```

The additional files generated by the compiler are:

- a `_XModule.java` class file for any IDL module X
- a `YIRHelper.java` class file for any interface Y.

If no `.class` files that are compiled from these extra classes are found in the class path passed to the IR server process, the IR will not be able to derive any representations. Note that the IDL compiler does not make any non–compliant modifications to any of the standard files that are defined in the Java language mapping — there is only additional information.

One more caveat about these extra classes: The compiler generates the `_XModule.java` class only for genuine modules. Java package scopes created by applying the `-d` switch to the IDL compiler do not represent proper modules and thus do not generate this class. Thus, the contents of these directories will not be considered by the IR.

When an object's client calls the `get_interface()` operation, the ORB consults the IR and returns an `InterfaceDef` object that describes the object's interface. Using `InterfaceDef` operations on this description object, further description objects can be obtained, such as descriptions for operations or attributes of the interface under consideration.

The IR can also be called like any other CORBA object and provides `lookup()` or `lookup_name()` operations to clients so that definitions can be searched for, given a qualified name. Moreover, the complete contents of individual containers (modules or interfaces) can be listed.

Interface Repository meta objects provide further description operations. For a given `InterfaceDef` object, we can inspect the different meta objects contained in this object (e.g., `OperationDef` objects). It is also possible to obtain descriptions in form of a simple structure of type `InterfaceDescription` or `FullInterfaceDescription`. Since structures are passed by value and a `FullInterfaceDescription` fully provides all contained descriptions, no further —possibly remote — invocations are necessary for searching the structure.

1.4 Interaction between #pragma prefix and -i2jpackage

Generally any use of *#pragma prefix* or *-i2jpackage* should be avoided if you intend to use an IDL file with the Interface Repository. If there is no other option there is a property that allows you to circumvent that restriction in some cases. Note however that this is a non-standard extension.

If, for example you have the following IDL file:

```
#pragma prefix "org.jacorb.test"

module ir
{
    typedef string StringAlias;
    typedef sequence<StringAlias> StringAliasList;

    struct TestStruct
    {
        StringAliasList stringList;
    };
};
```

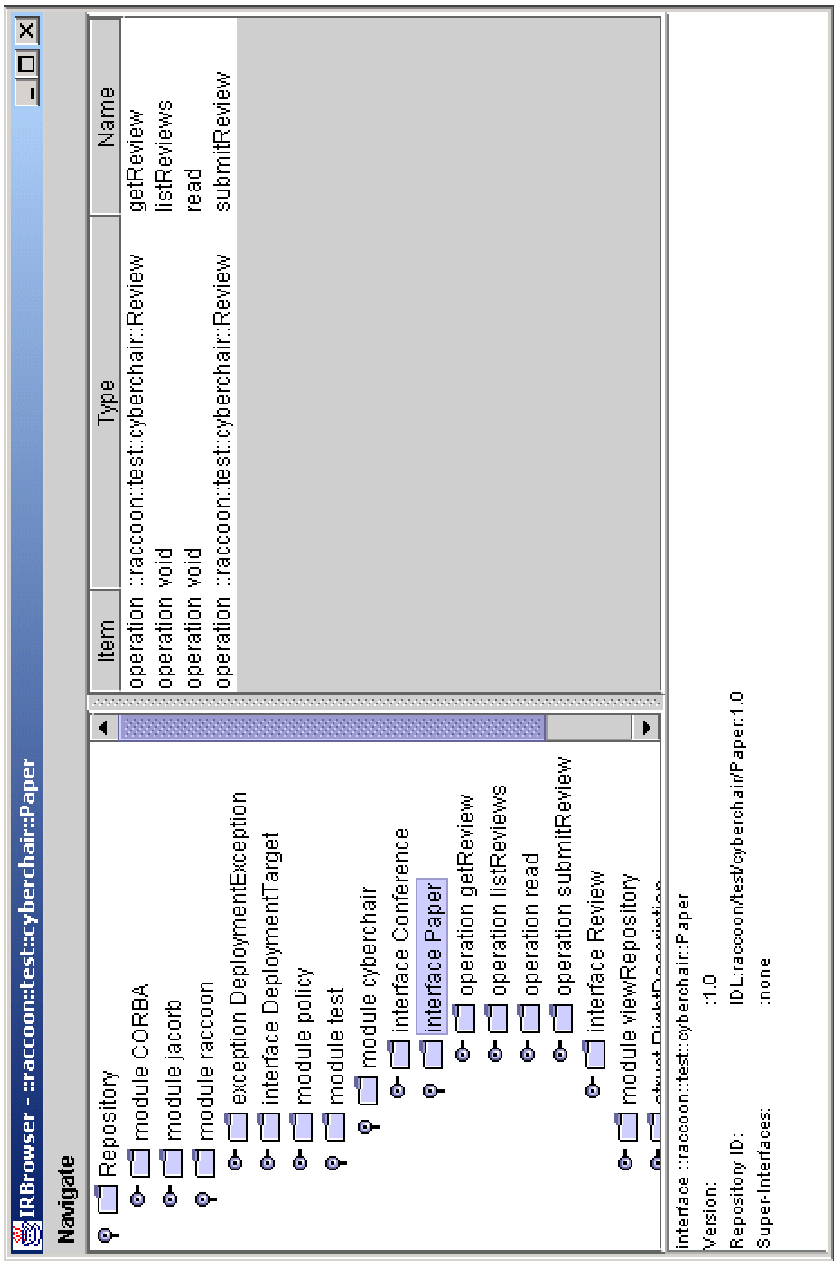
As you want your generated java files to reside in the package *org.jacorb.test.ir* you need to add *-i2jpackage* as an argument to the *idl* command. `$ idl -ir -i2jpackage ir:org.jacorb.test.ir myIdlFile.idl` Now the generated files are in the directory org/jacorb/test/ir.

As the IR starts it reads in the generated classes and implicitely creates their Repository ID's solely based on the directory structure. e.g. the struct TestStruct will get the Repository ID `IDL:org/jacorb/test/ir/TestStruct:1.0` however the correct Repository ID is `IDL:org.jacorb.test/ir/TestStruct:1.0`.

This will make it impossible for you to lookup the correct Repository ID successfully. starting of the IR will fail if the IR itself needs to look up a Repository ID during start.

As a workaround you can specify the property *jacorb.ir.patch_pragma_prefix=on* to the IR server. this property will cause the IR to change the first component of a requested Repository ID (Repository ID's consists of multiple components delimited with '/' so its org.jacorb.test in this case). If the first component looks like a pragma prefix (contains multiple '.') the '.' will be changed to '/'.

So the incoming request for `IDL:org.jacorb.test/ir/TestStruct:1.0` will be changed to a request for `IDL:org/jacorb/test/ir/TestStruct:1.0` so that the IR will be able to resolve that.



Item	Type	Name
operation	::raccoon::test::cyberchair::Review	getReview
operation	void	listReviews
operation	void	read
operation	::raccoon::test::cyberchair::Review	submitReview

interface ::raccoon::test::cyberchair::Paper
Version: :1.0
Repository ID: IDL:raccoon/test/cyberchair/Paper:1.0
Super-Interfaces: :none

Figure 1.3: IRBrowser Screenshot

2 JacORB Utilities

2.1 idl

The IDL compiler parses IDL files and maps type definitions to Java classes as specified by the OMG IDL/Java language mapping. For example, IDL interfaces are translated into Java interfaces, and typedefs, structs, const declarations etc. are mapped onto corresponding Java classes. Additionally, stubs and skeletons for all interface types in the IDL specification are generated.

(The IDL parser was generated with Scott Hudson's CUP parser generator. The LALR grammar for the CORBA IDL is in the file `org/jacorb/idl/parser.cup`.)

Compiler Options

-h | help	print help on compiler options
-v | version	print compiler version information
-d dir	root of directory tree for output (default: current directory)
-syntax	syntax check only, no code generation
-Dx	define preprocessor symbol x with value 1
-Dx=y	define preprocessor symbol x with value y
-Idir	set include path for idl files
-Usymbol	undefine preprocessor symbol
-W [1..4]	debug output level (default is 1)
-all	generate code for all IDL files, even included ones (default is off) If you want to make sure that for a given IDL no code will be generated even if this option is set, use the (proprietary) preprocessor directive `#pragma inhibit_code_generation`.
-forceOverwrite	generate Java code even if the IDL files have not changed since the last compiler run (default is off)
-ami_callback	generate AMI reply handlers and sendc methods (default is off). See chapter **??**
-ami_polling	generate AMI poller and sendp methods (default is off). See chapter **??**
-backend classname	use classname as compiler (code generator) backend. The default code generator class is `org.jacorb.idl.javamapping.JavaMappingGeneratingV` (c.f. API documentation). Custom generators must implement the interface `org.jacorb.idl.IDLTreeVisitor`

-i2jpackage x:a.b.c	replace IDL package name x by a.b.c in generated Java code (e.g. CORBA:org.omg.CORBA)
-i2jpackagefile filename	replace IDL package names using list from ¡filename¿. Format as above.
-ir	generate extra information required by the JacORB Interface Repository (One extra file for each IDL module, and another additional file per IDL interface (default is off)
-cldc10	Generate J2ME/CLDC1.0 compliant stubs
-genEnhanced	Generate stubs with toString/equals (only StructType)
-nofinal	generated Java code will contain no final class definitions, which is the default to allow for compiler optimizations.
-unchecked_narrow	use unchecked_narrow in generated code for IOR parameters in operations (default is off). Generated helper classes contain marshalling code which, by defa will try to narrow any object references to statically known interface type. This may involve remote invocations to test a remote object's type, thus incurring runtime overhead to achieve static type safety. The -unchecked_narrow option generates code that will not by statically type safe, but avoids remote tests of an object's type. If the type is not as expected, clients will experience CORBA.BAD_OPERATION exceptions at invocation time.
-noskel	disables generation of POA skeletons (e.g., for client-side use)
-nostub	disables generation of client stubs (for server-side use)
-diistub	generate DII-based client stubs (default is off)
-sloppy_forward	allow forward declarations without later definitions (useful only for separate compilation).
-sloppy_names	less strict checking of module name scoping (default: off) CORBA IDL has a number of name resolution rules that are stricter than necessary for Java (e.g., a struct member's name identifier must not equal the type name). The -sloppy_names option relaxes checking of these rules. Note that IDL accepted with this option will be rejected by other, conforma IDL compilers!
-sloppy_identifiers	permit illegal identifiers that differ in case (04-03-12:3.3.2) (default: off)
-permissive_rmic	tolerate dubious and buggy IDL generated by JDK's rmic stub generator (e.g., incorrectly empty inheritance clauses), includes -sloppy_names.
-generate_helper *compatibilty*	controls the compatibilty level of the generated helper code. Valid values are: **deprecated** uses CORBA 2.3 API. this API version is part of the JDK. **portable** uses CORBA 2.4 API. the usage of this option mandates the use of the JacORB provided *org.omg.** classes on the bootclasspath. This is the defau **jacorb** uses JacORB API. The generated helper code will contain references to JacORB classes. The helpers will use the CORBA 2.4 API but won't be portab anymore. There's no need to put the *org.omg.** classes provided by JacORB on the bootclasspath.

i2jpackage

The `-i2jpackage` switch can be used to flexibly redirect generated Java classes into packages. Using this option, any IDL scope x can be replaced by one (or more) Java packages y. Specifying `-i2jpackage X:a.b.c` will thus cause code generated for IDL definitions within a scope x to end up in a Java package `a.b.c`, e.g. an IDL identifier `X::Y::ident` will be mapped to `a.b.c.y.ident` in Java. It is also possible to specify a file containing these mappings using the `-i2jpackagefile` switch.

Example 1

given the following IDL definition

```
struct MyStruct
{
    long value;
};
```

Invoking idl without the i2jpackage option will generate (along with other files) the java file MyStruct.java

```
/**
 * Generated from IDL struct "MyStruct".
 *
 * @author JacORB IDL compiler V 2.3, 18-Aug-2006
 * @version generated at 07.12.2006 11:46:28
 */

public final class MyStruct
        implements org.omg.CORBA.portable.IDLEntity
{
    [...]
}
```

Note that the class does not contain a package definition.

The option -i2jpackage :com.acme will place any identifier without scope into the java package com.acme. Thus we get:

```
package com.acme;

/**
```

```
 * Generated from IDL struct "MyStruct".
 *
 * @author JacORB IDL compiler V 2.3, 18-Aug-2006
 * @version generated at 07.12.2006 11:46:28
 */

public final class MyStruct
        implements org.omg.CORBA.portable.IDLEntity
{
    [...]
}
```

Example 2

```
module outer
{
    struct OuterStruct
    {
        long value;
    };

    module inner
    {
        struct InnerStruct
        {
            long value;
        };
    };
};
```

If you're not using the i2jpackage option, the IDL compiler will generate the classes *outer.OuterStruct* and *outer.inner.InnerStruct*.

Again using the i2jpackage it's possible to map IDL modules to different java packages. `$ idl -i2jpackage outer:com.acme.outer` will generate the classes *com.acme.outer.OuterStruct* and *com.acme.outer.inner.InnerStruct*.

`$ idl -idjpackage inner:com.acme.inner` will generate the classes *outer.OuterStruct* and *outer.com.acme.inner.InnerStruct*.

Note: See Section 1.4 if you intend to use the i2jpackage option in conjunction with the JacORB IfR and are using #pragma prefix statements in your IDL.

Compiler Options

If one is building from Ant it is possible to invoke the compiler directly using the supplied Ant task, JacIDL. To add the taskdef add the following to the ant script:

```
<taskdef name="jacidl" classname="org.jacorb.idl.JacIDL"/>
```

The task supports all of the options of the IDL compiler.

Table 2.1: JacIDL Configuration

Attribute	Description	Required	Default
`srcdir`	Location of the IDL files	Yes	
`destdir`	Location of the generated java files	Yes	
`includes`	Comma-separated list of patterns of files that must be included; all files are included when omitted.	No	
`includesfile`	The name of a file that contains include patterns.	No	
`excludes`	Comma-separated list of patterns of files that must be excluded; files are excluded when omitted.	No	
`excludesfile`	The name of a file that contains include patterns.	No	
`defaultexcludes`	Indicates whether default excludes should be used (yes — no); default excludes are used when omitted.	No	
`includepath`	The path the idl compiler will use to search for included files.	No	
`parseonly`	Only perform syntax check without generating code.	No	False
`noskel`	Disables generation of POA skeletons	No	False
`nostub`	Disables generation of client stubs	No	False
`diistub`	Generate DII-based client stubs	No	False
`sloppyforward`	Allow forward declarations without later definitions	No	False
`sloppynames`	Less strict checking of names for backward compatibility	No	False
`generateir`	Generate information required by the Interface Repository	No	False
`all`	Generate code for all IDL files, even included ones	No	False
`nofinal`	Generate class definitions that are not final	No	False
`forceoverwrite`	Generate code even if IDL has not changed.	No	False
`uncheckedNarrow`	Use unchecked_narrow in generated code for IOR parameters in operations.	No	False
`ami`	Generate ami callbacks.	No	False
`debuglevel`	Set the debug level from 0 to 4.	No	0
`helpercompat`	control the portability of the generated helper code.	No	portable

Nested Elements

Several elements may be specified as nested elements. These are `<define>`, `<undefine>`, `<include>`, `<exclude>`, `<patternset>` and `<i2jpackage>`. The format of `<i2jpackage>` is `<i2jpackage names="x:y">`

Examples

The task command

```
<jacidl destdir="${generate}"
        srcdir="${idl}"
/>
```

compiles all *.idl files under the $idl directory and stores the .java files in the $generate directory.

```
<jacidl destdir="${generate}" srcdir="${idl}">
   <define key="GIOP_1_1" value="1"/>
</jacidl>
```

like above, but additionaly defines the symbol GIOP_1_1 and sets its (optional) value to 1.

```
<jacidl destdir="${generate}"
        srcdir="${idl}"
        excludes="**/*foo.idl"
/>
```

like the first example, but exclude all files which end with foo.idl.

2.2 ir

This command starts the JacORB Interface Repository, which is explained in chapter 1.

Usage

```
$ ir <reppository class path> <IOR filename>
```

2.3 qir

This command queries the JacORB Interface Repository and prints out re–generated IDL for the repository item denoted by the argument repository ID.

Usage

```
$ qir <reppository Id>
```

Bibliography